JavaScript for Modern Web Development

Building a Web Application Using HTML, CSS, and JavaScript

ALOK RANJAN
ABHILASHA SINHA
RANJIT BATTEWAD

bpb

JavaScript for Modern Web Development

FIRST EDITION 2020

Copyright © BPB Publications, India

ISBN: 978-93-89328-721

All Rights Reserved. No part of this publication may be reproduced or distributed in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's & publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners.

Distributors:

BPB PUBLICATIONS

20, Ansari Road, Darya Ganj

New Delhi-110002

Ph: 23254990/23254991

MICRO MEDIA

Shop No. 5, Mahendra Chambers,

150 DN Rd. Next to Capital Cinema,

V.T. (C.S.T.) Station, MUMBAI-400 001

Ph: 22078296/22078297

DECCAN AGENCIES

4-3-329, Bank Street,

Hyderabad-500195

Ph: 24756967/24756400

BPB BOOK CENTRE

376 Old Lajpat Rai Market,

Delhi-110006

Ph: 23861747

Published by Manish Jain for BPB Publications, 20 Ansari Road, Darya Ganj, New Delhi-110002 and Printed by him at Repro India Ltd, Mumbai

Dedicated to

Our Parents
who gave us roots to stay grounded and wings to fly high
Late Shri Arun Kumar Thakur and Smt. Nilu Thakur
Shri Ajit Kumar and Smt. Usha Sinha
Shri Narayan Battewad and Late Smt. Suwarna Battewad

About the Authors

Alok Ranjan is a B.Tech in Computer Science and Engineering from IIT Delhi. After graduating in 2001, Alok worked with companies like Newgen, Virtusa, and Convergys before starting his entrepreneurial journey. He co-founded an IT Services Company, Walking Tree, in 2008, which specializes in cross-platform web/mobile application development, digital transformation and data-driven decisions. He has been involved in defining and shaping technology products which have had a meaningful and measurable impact. When he is not working with customers, he spends time exploring technologies and building team.

Useful Links:

https://www.linkedin.com/in/alok-ranjan-b36a103/
https://walkingtree.tech/author/wtalok/
https://github.com/wtcalok

Abhilasha Sinha is BE Computer Science from Osmania University. After completing her graduation in 2003, she started her career as a Software Engineer with Infosys. She went on to be Senior Technology Architect in a long and fulfilling association of 12 years. She started as a Java developer, later moved to Oracle applications and also worked on middleware technologies like BPEL and OSB. She joined Walking Tree in 2016, where she continues to explore new technologies and deliver end-to-end custom web applications for enterprises using the latest technologies and frameworks like React.js, Angular and Node.js. She is also involved in training in Modern Web Frameworks like React and Angular. When not exploring something new, the mother of two boys loves spending time with her family.

Useful Links:

https://www.linkedin.com/in/abhilasha-sinha-0b795020/
https://skillgaze.com/
https://walkingtree.tech/author/abhilasha-sinha/

Ranjit Battewad has 9+ years of full-stack web and mobile application development experience. Associated with WalkingTree Technologies from last 8+ years and playing the role of a senior technical lead, he has exposure to complex application architecture design and development. His core expertise areas are Ext JS, Sencha Touch, HTML, CSS, JavaScript, NodeJS, MongoDB, PostgreSQL, Blockchain - Ethereum, Cordova, ReactJS,

Java. He has proven skills in developing and providing simple solutions for high complexity applications & development problems. He has a strong knowledge of databases, such as PostgreSQL and MySQL and is responsible for the complete life cycle of the project with exceptional project and team management skills.

Useful Links:

https://www.linkedin.com/in/ranjitbattewad/

https://walkingtree.tech/author/branjit/

https://github.com/ranjit-battewad

About the Reviewers

Suhail Abdul Rehman Chougule, Developed AI/ML Application, Linux, IBM Power AI Servers, Web Services, Apache Axis2 1.2, SOAP, Django, MS SQL Server, Ajax, Html, Multithreading,, LDAP (OpenDS), MySQL, CSS, Object-Oriented JavaScript, jQuery, jQuery UI, SVN, Circle CI (For Continuous Integration), Bugzilla, SSH. He Leads a team of 14 developers to develop and launch the AI-based Clinical App called Doctor App. Doctor App is a Web application and was implemented using MVC architectural pattern.

Suhail Abdul Rehman Chougule, Designed and implemented UAE based entities BI (Business Intelligence) dashboard which used Angular JS, Django, backend MS SQL Server deployed on CISCO Servers. He Evaluated new technologies. Added jQuery and jQuery UI as new technologies to be used in different projects. He Implemented a large part of the UI dynamic functionalities using CSS, jQuery, jQuery UI, Ajax, JSON, and XML. He Developed back end and front-end parts of Doctor App.

Acknowledgements

First and foremost, we would like to thank the Almighty for giving us the strength and capability to contribute to the field of knowledge. We would like to thank everyone at BPB Publications for giving us the opportunity to publish this book.

We would like to thank our parents and spouse for all the support system, so we were able to focus on the book, besides our regular work.

Special mention to our children Aayush, Akshar and Vaidik - their energy, positivity and curiosity, brightens our days and inspires us to keep going always.

Last but not least, we would like to thank Pradeep, Suman and the entire team of Walking Tree Technologies for all their timely support.

– Alok Ranjan

– Abhilasha Sinha

– Ranjit Battewad

Preface

In the last few decades, the web and the internet have grown by leaps and bounds, and it is now nothing short of omnipresent. JavaScript, the soul of the web and as old as the internet itself, still continues to grow and prosper. Numerous web and mobile JavaScript frameworks introduced to the world in the last few years, unleash the potential of JavaScript and make software application development, a very structured and easy to learn the process. The primary objective of this book is to create a strong foundation for web development by covering all aspects of it in one place. Starting from the basics of HTML, CSS and JavaScript, it brings about the different concepts of functional and object-oriented programming along with asynchronous constructs in JavaScript and goes on to introduce one of the popular JavaScript frameworks of today, React. This book includes step by step illustration of the development of two simple applications, using all the knowledge acquired here. This book also covers the supporting areas of the development process, which includes debugging, testing and deployment. This book will take you through the entire learning journey from Beginner to Expert supported with extensive code snippets, best practices, and concludes with applying the acquired knowledge to build real-life applications.

This book is divided into 18 chapters, and it will take you through the entire learning journey of Web application development from Beginner to Expert supported with code snippets and concludes with applying the acquired knowledge to build real-life applications.

Chapter 1 starts with the history of web development with Javascript, how it got started as a feeble companion and turned out to revolutionize the complete web development and how it continues to evolve and grow stronger and better.

Chapter 2 summarizes all the important HTML tags and how they can be used to put together the content for your web page. Starting with the basics, we will explore the new semantic tags and also different constructs like lists, links, forms, and images.

Chapter 3 introduces CSS, which will be used to beautify the content in numerous ways. It explores the power of CSS, understanding the CSS box model, positioning elements, and making your page responsive.

Chapter 4 discusses the basics of JavaScript, the multifaceted scripting language of the web, starting from variables to functions and event handling.

Chapter 5 discusses the functional programming paradigm which focuses on computing output rather than performing actions, i.e. focus on what to be done not on how and addresses functional programming using JavaScript

Chapter 6 describes the power of object-oriented programming(OOP), which enables you to start thinking as working with real-life entities or objects and understanding OOP in JavaScript.

Chapter 7 describes what is asynchronous flow and how it is achieved in JavaScript.

Chapter 8 introduces ECMA standards and discovers the new features(ES2019) and the essential changes of ES2015(ES6) of JavaScript with examples to put them to use in your applications.

Chapter 9 explains the process of building a full-fledged application from scratch, putting to use the knowledge of HTML, CSS and JavaScript acquired so far.

Chapter 10 describes how to debug, how to add logs, how to view the different DOM elements and track change of values in JavaScript applications.

Chapter 11 explains the need for Unit Testing and how it can be automated for web development using JavaScript testing frameworks like Jasmine.

Chapter 12 covers detailed steps to host your static application on the cloud platform so that your application is accessible over the internet using a URL.

Chapter 13 covers general JavaScript best practices to follow for any application development and some very useful tools which help in enforcing the standards.

Chapter 14 introduces the core concepts of React, a powerful JS library to build highly performant User Interfaces.

Chapter 15 explains how to approach a new application development using React and describes by building a real application using ReactJS.

Chapter 16 describes the flow of redux for state management and modify the React application to incorporate Redux for managing global state.

Chapter 17 explains how to debug, unit test and deploy React applications.

Chapter 18 introduces some additional topics based on various areas required to be proficient for taking the next leap to become a professional and building enterprise-level applications.

Downloading the code bundle and coloured images:

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/41bbjre

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors if any, occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Table of Contents

1. History of JavaScript and How it Revolutionized Web Development

- Structure
- Objective
- A brief history of the World Wide Web
- Web transition over the years
  - *Web 1.0*
  - *Web 2.0*
  - *Web 3.0*
- Advent of JavaScript
- ECMAScript – Standardization of JavaScript
- Role of JavaScript in Modern Web development
  - *Front-end JS frameworks*
  - *Back-end JS frameworks*
  - *Data layer frameworks*
  - *JS automation testing frameworks and test runner environments*
- Conclusion
- Questions

2. HTML - Creating the Web Content

- Structure
- Objective
- Getting started with HTML
- Building blocks of HTML
- Structure of an HTML document
- HTML elements
  - *Basic HTML tags to define content*
  - *Special content-related tags*
    - *HTML hyperlink*
  - *HTML list*
  - *HTML table*
    - *HTML form*
    - *Input form element*
    - *Other form elements*
- Special meaning-related HTML5 tags
    - *HTML comment tags*
    - *HTML images*
- HTML character entity references
- Conclusion
- Questions

3. CSS - Making the Content Beautiful
- Structure
- Objective
- Power of CSS
- Basics of CSS
- CSS syntax
  - *How to include CSS in HTML?*
    - *Inline style*
    - *Internal style sheet*
    - *External style sheets*
  - *Cascading styles*
  - *Working of CSS*
    - *Properties*
    - *Values*
    - *Selectors*
  - *Combinators*
    - *Contextual selectors/Descendant combinators*
    - *Child combinators/selector*
    - *Adjacent sibling selector*
    - *General sibling selector*
    - *Styling using Pseudo class*
    - *Styling with Pseudo-elements*
    - *CSS inheritance*
- CSS precedence and specificity
  - *CSS precedence rules*
    - *Use of !important after CSS properties*
    - *Specificity of CSS rule selectors*
- Sequence of declaration
  - *CSS units*
    - *Absolute lengths*
    - *Viewport lengths*
    - *Font-relative lengths*
    - *Percentage*
- CSS styling – Font, text, background properties
  - *Styling font*
  - *Styling text*
  - *Styling background*
- CSS – The box model
  - *Box-model*
  - *Setting width and height*
    - *Box sizing*
    - *Vertical margin collapse*
  - *CSS display property*
    - *Inline box*

Block box
Inline-block
CSS overflow
Positioning elements
Fixed
Absolute
Relative
Sticky
Stacking context
CSS Flexbox
Flex container properties
Flex-item properties
Responsive design
Viewport
Media queries
Conclusion
Questions

4. JavaScript Programming: Making the Application Interactive
Structure
Objective
Introduction to JavaScript (JS)
Building blocks of JS
Variables
Scopes
Global scope
Local scope
Data Types
Working with Data Types
Operators
Types of operators:
Comments
Statements
Functions
What is a function?
More about functions
Function as expression
Function declaration versus function expression
Different aspects of functions
Arrays
Working with DOM
Traversing the DOM
Nodes related methods
Element related methods

Accessing elements
Getting and updating element content
Adding and removing HTML content
Attribute related
Conclusion
Questions

5. Functional Programming with JavaScript
Structure
Objective
Functional programming
Imperative style
Declarative style/functional
Features of functional programming
Using functional programming
Major concepts of functional programming
Pure functions
Benefits of pure functions
Higher-order functions
Immutability
Recursion
Referential transparency
First-class functions
Stored in a variable
Passed as an argument to any other function
Returned by any other function
Limitations of a functional programming language
Conclusion
Questions

6. Object-Oriented JavaScript
Structure
Objective
Introduction to OOP
Class
The constructor of a class
Static member in a class
Object
Class expression
Encapsulation
Inheritance
Use of super inside the constructor
Method override
Calling the parent function

Inheriting from built-in objects
Abstraction
Conclusion
Questions

7. Asynchronous JS
Structure
Objective
Asynchronicity in JavaScript
JavaScript runtime
Callbacks
Promises
Async Await
Async
Await
Scenario 1 – Synchronous behavior within asynchronous
Scenario 2 – Parallel execution within asynchronous
Scenario 3 – Async with classes
Scenario 4 – Error handling for async-await
Scenario 5 – Working with arrays asynchronously
Conclusion
Questions

8. What's New in ES2019 JavaScript
Structure
Objective
ECMA Script Standardization
Essential ES6 changes
Handling variables from var to const/let
The power of arrow functions
Handling strings
Template literals
Handling parameters
From arguments to rest parameters
Symbols
Spread operator
Destructuring
Classes
ES2019 JavaScript – Major new features
Array.prototype.{flat,flatMap}
Array.flat()
Array.flatMap()
Object.fromEntries
ES2019 JavaScript – Minor new features

String.prototype.{trimStart,trimEnd}
String.trimStart()
String.trimEnd()
Symbol.prototype.description
Optional catch binding
New Function.toString()
JSON ⊂ ECMAScript (JSON Superset)
Well-formed JSON.stringify()
Stable Array.prototype.sort()
Conclusion
Questions

9. Building An Application with JavaScript
Structure
Objective
Planning the application
Solution approach
Building the application step-by-step
Step 1 – Adding content to the launch page
Step 2 – Styling the launch page
Step 3 – Logic for the launch page
Step 4 – Adding responsiveness to the launch page
Step 5 – Content and styling for the application page
Step 6 – Content and styling for notes list in the application page
Step 7 – Logic for the application page
Conclusion
Questions

10. Debugging JavaScript Applications
Structure
Objective
The browser devtools
How to launch the browser devtools?
Parts of the Chrome devtools
Elements and styles
Console
Sources
Network
Application
Performance
Memory
Security
Audit
The debugging process

Using console.log statements
Using the JavaScript Debugger
How to add breakpoints?
Conclusion
Questions

11. Unit Testing Automation
Structure
Objective
Unit testing
Introduction to Jasmine
Jasmine setup and configuration
Jasmine basic constructs
describe
it
beforeEach and afterEach
expect
Matchers
Asynchronous testing
Conclusion
Questions

12. Build and Deploy an Application
Structure
Objective
Why deployment?
Getting ready for deployment
For HTML/CSS/JS applications
Detailed steps for hosting
Step 1 – Login to the Heroku CLI
Step 2 – Initialize and commit the application code to Git
Step 3 – Create a named app in Heroku CLI
Step 4 – Push your code into the remote git repository
Step 5 – Deployment of any changes made to the application
Alternate method
Conclusion
Questions

13. JavaScript Best Practices
Structure
Objective
JavaScript best practices
Readability
Naming conventions

Indentation and spacing
Guidelines for objects
Add sufficient comments as needed
Avoid global variables
Follow strict coding style
Write modular code – Single responsibility principle
Always declare variables
Always initialize variables
Avoid heavy nesting
Optimize loop logic
Use = = = Comparison
Differentiate between assignment and comparison
Differentiate between null and undefined
Always use semicolon
Use default in switch statement
Differentiate between addition and concatenation
Trailing comma in JSON and object definitions
HTML and CSS best practices
Maintain a clean and readable code structure
Include JS Script at the end
Avoid inline styling
CSS to be used consciously
Include imports wisely
Use lists for navigation items
Avoid unnecessary divs
Avoid tag qualify
Use color hex codes
Consider cross-browser compatibility
Validate your code
Reduce CSS code size
Recommendations on best practices
Conclusion
Questions

14. Introduction to React
Structure
Objective
Single Page Applications(SPA)
Getting started with React
src
React and React-DOM
Introduction to JSX
Adding event handlers
Elements, components and props

Elements
Components
Functional components
Class components
Stateful components and stateless components
Props
PropTypes
Passing method references between components using Props
State, lifecycle and virtual DOM
State
Manipulating state
Asynchronous setState function
Lifecycle methods
Mounting
Updating
Unmounting
Virtual DOM
Styling components
Styling using external CSS
Inline styling
Rendering lists and conditionals
Lists
Conditional rendering
Forms
Uncontrolled forms
Controlled forms
Composition
Hooks
The useState hook
The useEffect hook
How can hooks be reused?
Conclusion
Questions

15. Building an Application with React
Structure
Objective
Thinking in React to Approach the UI Development
Requirements and Application Design
Prerequisites and getting started
Building the React application
Step 1
Step 2
Step 3

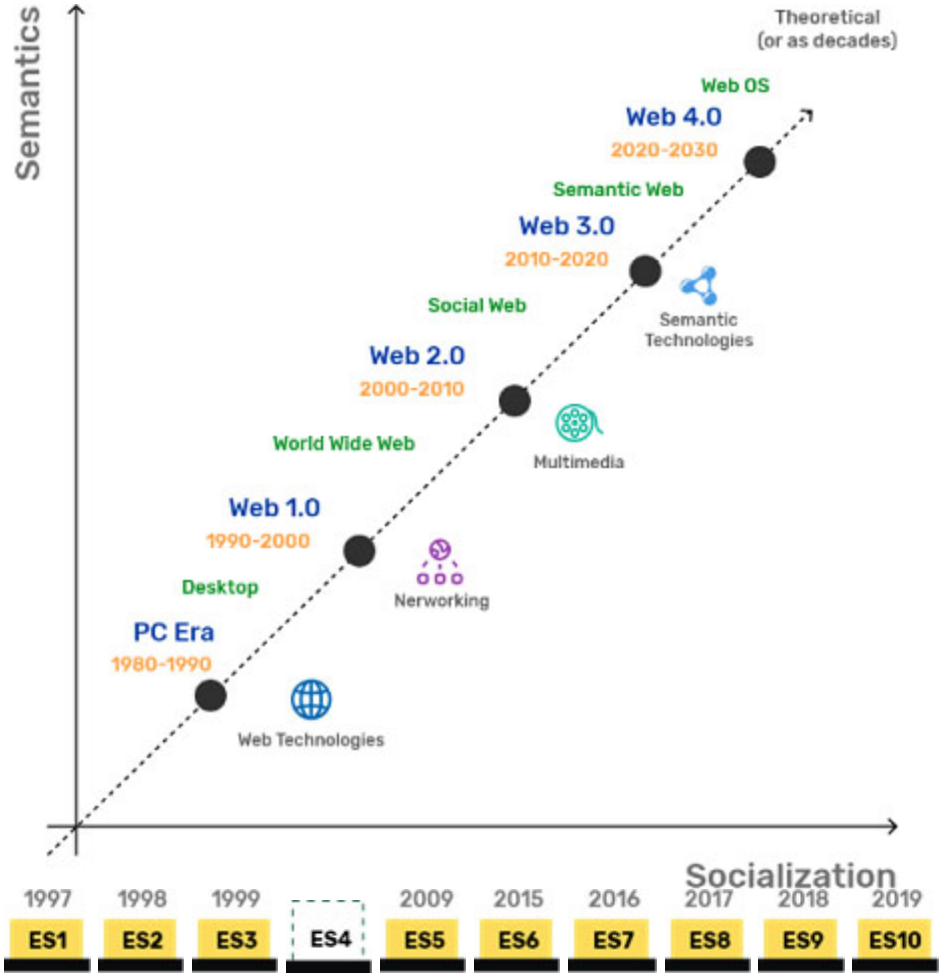
Step 4
Step 5
Step 6
Step 7
Step 8
Conclusion
Questions

16. State Management in React Applications
Structure
Objective
State management
The architecture of Redux
Why is middleware needed?
Incorporating Redux in React
Set up and provide the Redux store to the React application
React components connect to the Redux store
Conclusion
Questions

17. Debugging, Testing, and Deploying React
Structure
Objective
Debugging React applications
Reactdeveloper tools
Redux devtools
Testing React applications
Test file organization
Defining the tests
Snapshot testing
Shallow and mount
Deploying React applications
Conclusion
Questions

18. What Next - For Becoming A Pro?
Structure
Objective
Modern web development with JavaScript frameworks
Building mobile apps
Templating engines
Visualization
Functional programming
Routing

Logging
Internationalization
Documentation
Testing frameworks
QA tools
Package manager
Code coverage
Runner
Services
Database
Conclusion
Questions



CHAPTER 1

History of JavaScript and How it Revolutionized Web Development

"We are all now connected by the Internet, like neurons in a giant brain."

— Stephen Hawking, one of the most influential scientists of this era.

From the initial hesitant reception of the first proposal of the web to more than 1 billion websites on the internet today, the web has grown

exponentially. It continues to grow with 4.1 billion Internet users in the world as of December 2018. JavaScript, the simple scripting language which came into existence to be able to add dynamicity and interactiveness to the web quickly, rose from very humble beginnings to become a powerful language and continues to grow in its features. The power of JavaScript is the real reason behind the growth of the web and its capabilities, because of the ease with which JavaScript facilitates modern web development for both client-side as well as server-side. Be it pure vanilla JavaScript or the JavaScript in the form of frameworks like React and Node, and JavaScript has become next to irreplaceable in the world of web development.

Structure

- A brief history of the World Wide Web
- Web transition over the years
- Advent of JavaScript
- ECMAScript –Standardization of JavaScript
- Role of JavaScript in modern web development

Objective

After studying this chapter, you will get an understanding of the history of the web and JavaScript and also how it has grown in terms of new features as well as in the form of various frameworks and libraries.

A brief history of the World Wide Web

In March 1989, Sir Tim Berners-Lee anticipated the need for an information management system to be able to share data globally and submitted a proposal for the same. The response of his boss Mike Sendall was 'Vague, but exciting' and he got the approval to proceed. He designed a new way of accessing the information on other computers and called his creation, the World Wide Web.

By October 1990, Tim had written the three fundamental technologies that formed the rock-solid foundation of the world wide web:

- **HyperTextMarkup Language (HTML):** It is the markup language for creating content on the web.
- **Uniform Resource Identifier (URI):** It is the unique address used to identify each resource on the web, also commonly called a **URL(Uniform Resource Locator).**
- **HyperText Transfer Protocol (HTTP):** It is the protocol for access to

linked resources from across the web.

From **http://info.cern.ch/,** the address of the world's first website and Web server, running on a NeXT computer at CERN, the web continued to grow with simple line-mode browsers to Mosaic 1.0 being the first web browser to allow images and text to load on the same page. Netscape Navigator was launched in 1994, which became a huge hit.

Sir Tim Berners-Lee founded the **World Wide Web Consortium (W3C),** to make sure the web stays free and accessible to all. His vision of the World Wide Web was of an open, universal space, where anyone and everyone was free to express their ideas and showcase their creativity. This freedom initiated an astounding transformation in the world, bringing out the best of innovation that has changed the world in the last three decades.

Web transition over the years

Over the years, the web has seen major changes, starting from the mostly read-only, shareable web1.0 to socially enabling web 2.0 to highly intelligent and powerful 3.0, and it continues to evolve and grow:

Figure 1.1: *Web Transition over the years*
(Image reference: Radar Networks and Nova Spivac 2007, www.radarnetworks.com)

Web 1.0

You have content to share? Share it!

It started with Web 1.0, the first phase of web development, the read-only web where the content was mostly static with hyperlinks to view the content and bookmarking to save the links. The emails were sent through HTML forms.

It was mostly to share content over the web with minimal user interaction or content generation during the interaction. The content of the website was stored in files on the server from where the data was retrieved and rendered on the client on receiving a request. The comments given by the visitors were added to a common Guestbook page rather than to the direct content to avoid slowing down the website. The users could search for content and read it, so it is sometimes referred to as the "read-only web".

Web 2.0

Do you have an opinion? Speak up!

With the arrival of databases, upgraded servers, improved connection speeds, the static Web1.0 transitioned to the interactive Web 2.0. Now the user was also creating live content on the web through comments, tweets, videos, and posts. This phase saw the advent and growth of social media

giants like Facebook and Twitter, video streaming website like YouTube. The focus shifted on user experience as the user was also a participant in creating web content. Also, there was added capability to include content from one website into some other external website through the use of **APIs (application programming interfaces)**.

The web was becoming more and more powerful with the capabilities at hand being limitless. What came next was nothing less than miraculous!

Web 3.0

You don't need to live with inefficiencies! You get what you ask for!

The third generation web can be coined as the **intelligent web** comprising of semantic web technologies, distributed databases, distributed computing, micro-formats, natural language processing, data mining, machine learning and reasoning, recommendation agents, and artificial intelligence technologies. Now the machines are empowered to understand and decode the information and provide a more intuitive user experience where they understand exactly what the user has searched for.

What lies beyond is the extent of where our imagination and innovation can take us!

Advent of JavaScript

JavaScript was created by Brendan Eich in 1995 during his association with Netscape Communications.

Netscape was the most popular browser at those times, but it was being threatened by the arrival of Microsoft's Internet Explorer. To combat this, Netscape partnered with Sun so that they could be the official browser of the much-awaited Sun's platform – Java, and maintain their popularity.

JavaScript came out as a result of the need to make the web dynamic. It was positioned as an easy to use language which even the Web designers and part-time developers could use to put together components such as images and plugins directly into the Web page markup.

It was developed under the name Mocha, first shipped as LiveScript and later changed to JavaScript just to position it as a companion to Java and benefit from the ongoing marketing value of the hot programming language.

Bui-in a rush to be placed as Java's companion, there were many features of JavaScript which were disliked by the users for being a deviation from the other programming languages of that time which the users were habituated to use.

Some of such features include:

- AutomaticSemicolon Insertion (ASI)
- Automatic type coercion when using common operators like '= ='
- Lack of block scoping
- Lack of classes
- Lack of dedicated modularization capability
- Unusual inheritance (prototypical)

Netscape released JavaScript for browsers in December 1995,and also introduced an implementation of the language for server-side scripting.

From the very humble beginnings of being a companion to becoming the de-facto standard language of the web, JavaScript has come a really long way.

Despite the criticisms and multiple attempts by huge companies to suppress and replace, JavaScript continues to grow, as it is open, standardized, and a powerful language tightly coupled with the DOM and the best way to make dynamic web content.

ECMAScript – Standardization of JavaScript

One of the most significant events in the history of JavaScript was ECMA standardization. ECMA is an industry body founded in 1961, which deals with the standardization of information and communication systems.

By the time the standardization first started for JavaScript in November 1996, it was already in full use, by an estimated 3 lakh pages as per a Netscape press release. The standard identification was ECMA-262, and the committee responsible for the standardization was TC-39.

This standardization ensured controlled implementation and proper evolution of the language over all these years without any fear of deviation and fragmentation.

The ECMA committee was not able to retain JavaScript as the name due to trademark issues. The committee could not agree to like any of the alternative names, so after some deliberation and discussion, it was decided that the language described by the standard would be called ECMAScript. Today, JavaScript is just the commercial name for ECMAScript.

The first two versions ECMAScript 1 & 2 were focused on taking the original version towards standardization. The ECMAScript 3, released in December 1999, saw the first set of major changes.

The same year also witnessed the birth of AJAX (asynchronous JavaScript and XML) by Microsoft as the `XMLHttpRequest` function which allowed a browser to perform an asynchronous HTTP request against a server, thus allowing pages to be able to fetch data in the background and get updated

while you continue interacting with the page.

The next version, ECMAScript 4, which proposed a huge set of features, underwent a lot of opposition and discussions for 8 long years due to the difference of opinions of the parties involved. Meanwhile, there was a parallel version being worked upon named as ECMAScript 3.1 with some agreeable set of features. ECMAScript 4 was finally scrapped, and ECMAScript 3.1 renamed as ECMAScript 5 became one of the most supported versions of JavaScript. ECMAScript 5 included some important set of features which gave shape to the future of JavaScript.

Next came ECMAScript 6 (2015) and 7 (2016) which added many awaited features are establishing JavaScript as a powerful programming language.

ECMAScript 6 (2015) was another major version which has received widespread acceptance. Next came ECMAScript 7 (2016) which included a rather smaller set of features.

The subsequent years have seen new features added as ECMAScript 8 (2017), ECMAScript 9 (2018), ECMAScript 10 (2019).

JavaScript continues to evolve with each version of ECMAScript, further strengthened to maintain its hold on the present as well as the future of web development.

Table 1.1 shows a summary of the features of the different ECMAScript versions which have evolved year after year, and continue to grow and bring new capabilities to the JavaScript language:

The standardization of the original version

The standardization of the original version

- Regular expressions
- The `do…while` block
- Exceptions and `try…catch` blocks
- More built-in functions for strings and arrays
- Formatting for numeric output
- The in and `instanceof` operators
- Much better error handling

Too many features, after much deliberation, got scrapped

- Getter/setters
- Trailing commas in array and object literals
- Reserved words as property names
- New Object, Array and Date methods
- `String.prototype.trim` and property access

- Function bind
- JSON
- Immutable global objects (undefined, NaN, Infinity)
- Strict mode
- Let (lexical) and const bindings
- Arrow functions (shorter anonymous functions) and lexical this (enclosing scope this)
- Classes (syntactic sugar on top of prototypes)
- Object literal improvements (computed keys, shorter method definitions, and so on)
- Template strings
- Promises
- Generators, iterables, iterators and for…of
- Default arguments for functions and the rest operator
- Spread syntax
- Destructuring
- Module syntax
- New collections (`Set, Map, WeakSet, WeakMap`)
- Proxies and reflection
- Symbols andtyped arrays
- Support for sub classing built-ins
- Guaranteed tail-call optimization
- Simpler unicodesupport
- Binary and octal literals
- The exponentiation operator (**)
- `Array.prototype.includes` a few minor corrections (generators can't be used with new, and so on)
- `Object.entries`
- `Object.values`
- String padding
- `Object.getOwnPropertyDescriptors`
- Trailing commas in function parameter lists and calls
- Async functions
- Exponentiation operator
- `Array.prototype.includes`

- Asynchronous iteration
- Rest/Spread properties
- New regular expression features
- `Promise.prototype.finally()`
- Template Literal Revision
- `Array#{flat,flatMap}`
- `Object.fromEntries`
- `String#{trimStart,trimEnd}`
- `Symbol#description`
- `try { } catch {}` // optional binding
- Subsume JSON (JSON ⊂ ECMAScript)
- Well-formed `JSON.stringify`
- Stable `Array#sort`
- Revised `Function#toString`

Table 1.1: *ECMAScript Versions with main features*

Role of JavaScript in Modern Web development

From static read-only web to the highly interactive web of today, JavaScript is the reason behind this incredible transformation. While HTML defines the content, CSS defines the look and feel, JavaScript is responsible for defining the behaviour of an application. It is what makes the web dynamic and interactive. JavaScript is capable of manipulating the content part(HTML) as well as the presentation part(CSS), and hence, it is the powerful master of the web.

JavaScript dominates the complete web market in the form of the wide variety of powerful frameworks and libraries which make the web development process very well-structured and easy. JavaScript is omnipresent on the web, be it the client-side code or the server-side logic, it handles all.

The feature-rich libraries and frameworks like Angular, React, make web development easier by enabling developers to add functionalities without writing any complex code. JavaScript frameworks provide simplified structure to complex commands in the form of simple blocks of JavaScript code, thereby making the process of programming easier and faster.

Today the process of software development is primarily composition: breaking down complex problems into smaller problems, and solving the smaller problems which are finally combined to get the complete solution in the form of your application. All the modern frameworks facilitate development following the composition approach wherein the complete

design is broken down into smaller components, which are constructed piece by piece and put together to build the solution as a whole.

The list of JS Frameworks is ever-growing and evolving, which further strengthens its presence on the web. Following is just a fragment consisting of the popular JS frameworks and libraries.

Front-end JS frameworks

There are many popular front end frameworks which have madethe entire process of web development very easy. It includes:

- **React:** Facebook's popular and flexible JavaScript library, which is very powerful for building web application user interfaces
- **Angular:** A JavaScript-based open-source front-end web framework backed by Google which has simplified the development process of single-page applications
- **Vue:** Another open-source JavaScript framework is known for its flexibility and simplicity in building single-page applications

Back-end JS frameworks

From JavaScript runtime environments to server-side frameworks, JS frameworks enabling the back end layer include:

- **Node.js:** A server-side JavaScript runtime environment built on Chrome's V8 JavaScript engine, designed to make use of JavaScript to build back-end applications.
- **Express:** This popular server-side JavaScript framework which runs on the Node.js platform, helps build websites, web application servers with ease.
- **Meteor.js:** Another open-source, real-time backend framework, built on top of Node.js and works with MongoDB, a popular NoSQL open-source database (**https://www.upwork.com/hiring/data/should-you-use-mongodb-a-look-at-the-leading-nosql-database/**).

Data layer frameworks

Some popular JS frameworks which facilitate managing and accessing the application data include:

- GraphQL (**https://graphql.org/**) to query database services
- Redux (**https://redux.js.org/**) is used to manage application state for JS applications

JS automation testing frameworks and test runner

environments

Not only web development, but these JS Frameworks also enable testing by providing automation testing frameworks and test runner environments:

- Jasmine
- Mocha
- Jest
- Karma
- TestCafe

Other JavaScript frameworks and technologies to look out for:

Some other popular JavaScript frameworks directly used in web development or facilitating some other aspect of the development:

- Ext JS
- Backbone.js
- Ember.js
- Socket

From beautiful and dynamic front ends to fast and efficient backends, from accessing databases to automating test cases; JavaScript is all-over the web in the form of these powerful frameworks providing numerous capabilities to the entire web development process.

Conclusion

This book is the beginning of a fulfilling journey into the world of web development. The world of web is boundless, and there is really no limit to what you can achieve. Starting from the basics, we will embark on this amazing path and empower you with the foundation tools. With a strong foundation of fundamentals, you can move ahead with continuous learning and practice and surely build a superstructure. In the next chapter, you will start your learning with HTML, the language for defining the content of the web.

Questions

1. Which major version of ECMAScript included the compact arrow function, the syntactic sugar in the form of classes, the destructuring of objects and arrays, and so on?

    A. ES5

    B. ES6

C. ES7

D. ES8

Answer: Option B.

ES6(2015) saw the inclusion of the useful features of arrow function, class, destructuring, promises, default arguments for functions, and so on.

2. Which version of the web is considered to have introduced socializing on the web by creating live content in the form of comments, tweets, videos, and so on?

   A. Web 1.0

   B. Web 2.0

   C. Web 3.0

   D. Web 4.0

   Answer: Option B.

   Web 2.0 version was the arrival of social media, sharing live data feed, creating live content in various forms like comments, tweet, video, audio, and so on.

3. Which of these is the JS framework which helps in state management of applications?

   A. Jasmine

   B. Redux

   C. Backbone

   D. Express

   Answer: Option B.

   Redux (**https://redux.js.org/**) is used to manage application state for JS applications.

4. Which of these technologies forms the foundation of the worldwide web?

   A. HTML: HyperTextMarkup Language

   B. URI: Uniform Resource Identifier.

   C. HTTP: Hypertext Transfer Protocol.

   D. All of the above

   Answer: Option D.

   The three fundamental technologies that formed the rock-solid foundation of the worldwide web include:

   - **HyperText Markup Language (HTTP):** The markup language

for creating content on the web.

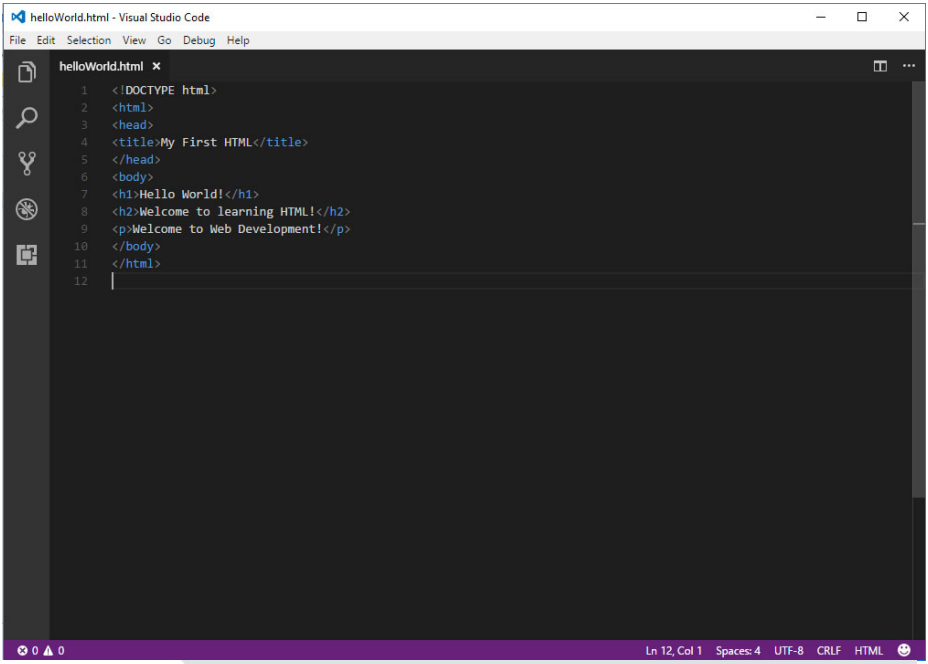
- **Uniform Resource Identifier (URI):** The unique address used to identify each resource on the web, also commonly called a URL.
- **Hypertext Transfer Protocol (HTTP):** The protocol for access to linked resources from across the web.

8. Breaking down complex problems into smaller problems, and solving the smaller problems which are finally combined to get the complete solution in the form of your application. What is this approach called?

   A. Composition
   B. Consolidation
   C. Reconciliation
   D. Development

   Answer: Option A.

   The process of breaking down complex problems into smaller problems, and solving the smaller problems which are finally combined to get the complete solution in the form of your application, is called composition.

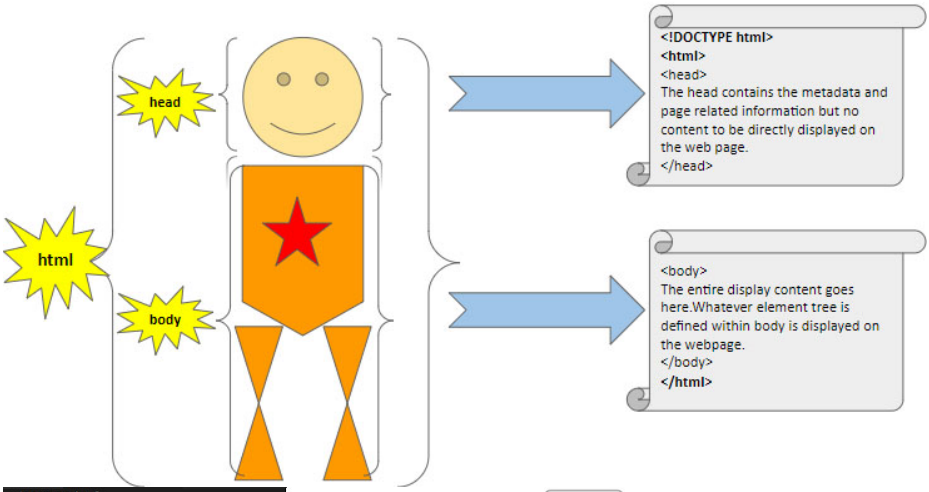


Hello World!

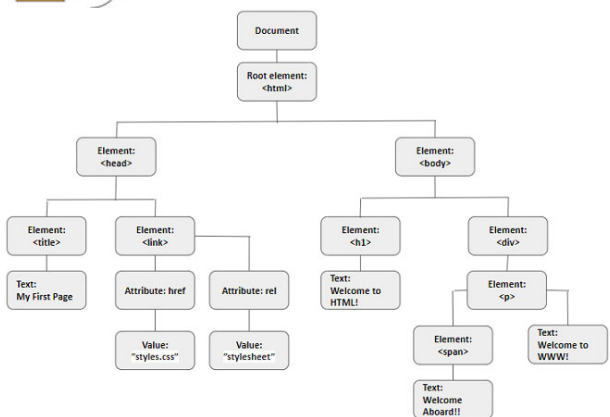
Welcome to learning HTML!

Welcome to Web Development!

WEB PAGE STRUCTURE




```
<!DOCTYPE html>
<html>
<head>
<title>My First HTML</title>
<link rel="stylesheet" href="styles.css"/>
</head>
<body>
<h1>Welcome to HTML!</h1>
<div>
<p>Welcome to WWW!
    <span> Welcome Aboard!! </span>
</p>
</div>
</body>
</html>
```




```
<!DOCTYPE html>
<html>
<head>
<title>My First HTML</title>
</head>
<body>
<h1>Hello World!</h1>
<h2>Welcome to learning HTML!</h2>
<div>
<p>Welcome to Web Development!
<span> Welcome Aboard!! </span>
</p>
<p> This is an exciting world !
 <span> Have Fun </span>
</p>
</div>
</body>
</html>
```

Hello World!

Welcome to learning HTML!

Welcome to Web Development! Welcome Aboard!!

This is an exciting world ! Have Fun

```html
<!DOCTYPE html>
<html>
<head>
    <title>My First HTML with basic tags</title>
</head>
<body>
    <h1>Level 1 Heading</h1>
    <h2>Level 2 Heading</h2>
    <h3>Level 3 Heading</h3>
    <h4>Level 4 Heading</h4>
    <h5>Level 5 Heading</h5>
    <h6>Level 6 Heading</h6>
 <div>
        <p>Welcome          to          Web          Development!
         This is an exciting world !<br>
         The spaces    and
        new lines are not considered in a paragraph tag
         </p>
<pre> The spaces, new lines, fonts will be retained in the pre tag!
    <p> This is what different formats will look like
        <strong> strong text </strong>, <b> bold text </b>,
        <i> Italized with i</i>, <em> emphasized with em</em>,
          <small> small in size</small>,
          <sub> subscript </sub> or <sup> superscript </sup>,
          <ins> Added </ins> or  <del> Deleted</del>, or
          <mark> Finally highlighted</mark>
    </p>
</pre>
</div>
</body>
</html>
```

My First HTML with basic tags

File | C:/xampp/htdocs/JSBook/HTML/helloTags.html

Level 1 Heading

Level 2 Heading

Level 3 Heading

Level 4 Heading

Level 5 Heading

Level 6 Heading

Welcome to Web Development! This is an exciting world !
The spaces and new lines are not considered in a paragraph tag

```
 The spaces, new lines, fonts will be retained in the pre tag!
```

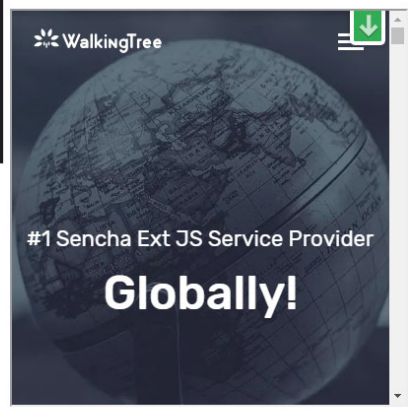
This is what different formats will look like
strong text , bold text ,
Italized with i, emphasized with em,
small in size,
subscript or superscript ,
Added or ~~Deleted~~, or
Finally highlighted

```html
<!DOCTYPE html>
<html>
<head>
<title>HTML Links</title>
</head>
<body>
<h1>Hello World!</h1>
<h2>Welcome to learning HTML!</h2>
<div>
<p>Welcome to Web Development!<span> Welcome Aboard!! </span></p>
<iframe name="test" height="400px" width="400px"></iframe><br>
<a href="https://walkingtree.tech" target="test" ><br>
 Take me to Walkingtree</a><br>
</div>
</body>
</html>
```

Hello World!

Welcome to learning HTML!

Welcome to Web Development! Welcome Aboard!!



Take me to Walkingtree

htmlList.html

```html
<!DOCTYPE html>
<html>
<head>
<title>My First List</title>
</head>
<body>
<ul>
<li> Home </li>
<li> Contact Us </li>
<li> Dashboard </li>
<li> Customers </li>
</ul>
</body>
</html>
```

My First List

file:///C:/JSBook/HTML/htmlList.html

- Home
- Contact Us
- Dashboard
- Customers

htmlList.html

```html
<!DOCTYPE html>
<html>
<head>
<title>My First List</title>
</head>
<body>
<ol>
<li> Home </li>
<li> Contact Us </li>
<li> Dashboard </li>
<li> Customers </li>
</ol>
</body>
</html>
```

My First List

file:///C:/JSBook/HTML/htmlList.html

1. Home
2. Contact Us
3. Dashboard
4. Customers

```html
<body>
<ol type="I">
<li> Home </li>
<li> Contact Us </li>
<li> Dashboard
    <ul style="list-style-type: square">
        <li> Daily Report </li>
        <li> Weekly Report </li>
        <li> Monthly Report </li>
    </ul>
</li>
<li> Customers
    <ul >
        <li> Customer Relations</li>
        <li> Customer Support </li>
        <li> Customer Escalations </li>
    </ul>
</li>
</li>
</ol>
</body>
```

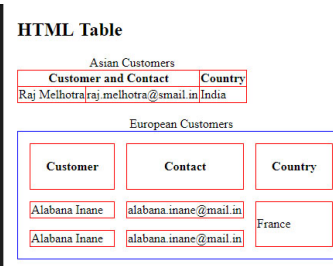
I. Home
II. Contact Us
III. Dashboard
    - Daily Report
    - Weekly Report
    - Monthly Report
IV. Customers
    - Customer Relations
    - Customer Support
    - Customer Escalations

```html
<table>
  <tr>
    <th>Customer</th>
    <th>Contact</th>
    <th>Country</th>
    <th>Feedback</th>
  </tr>
  <tr>
    <td>Alabana Inane</td>
    <td>alabana.inane@mail.in</td>
    <td>France</td>
    <td>Good</td>
  </tr>
  <tr>
    <td>Raj Melhotra</td>
    <td>raj.melhotra@smail.in</td>
    <td>India</td>
    <td>Good</td>
  </tr>
</table>
```

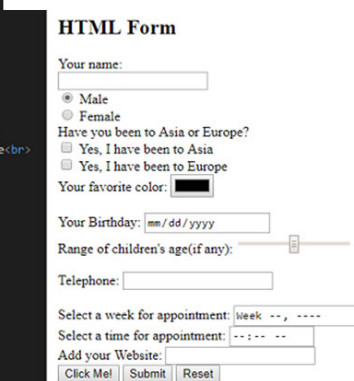
HTML Table

Customer	Contact	Country	Feedback
Alabana Inane	alabana.inane@mail.in	France	Good
Raj Melhotra	raj.melhotra@smail.in	India	Good

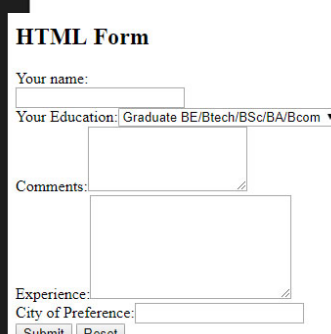
```
<h2>HTML Table</h2>
<table style="border:1px solid blue;  border-collapse: collapse; border-spacing:15px">
    <caption>Asian Customers</caption>
  <tr>
    <th style="border:1px solid red;" colspan="2">Customer and Contact</th>
    <!--<th style="border:1px solid red">Contact</th>-->
    <th style="border:1px solid red">Country</th>
  </tr>
  <tr>
    <td style="border:1px solid red">Raj Melhotra</td>
    <td style="border:1px solid red">raj.melhotra@smail.in</td>
    <td style="border:1px solid red">India</td>
  </tr>
  </table>
  <br/>
<table style="border:1px solid blue;  border-collapse: separate; border-spacing:15px">
  <tr>
    <th style="border:1px solid red; padding:20px">Customer</th>
    <th style="border:1px solid red; padding:20px">Contact</th>
    <th style="border:1px solid red; padding:20px">Country</th>
  </tr>
  <tr>
    <td style="border:1px solid red">Alabana Inane</td>
    <td style="border:1px solid red">alabana.inane@mail.in</td>
    <td style="border:1px solid red" rowspan="2">France</td>
  </tr>
   <tr>
    <td style="border:1px solid red">Alabana Inane</td>
    <td style="border:1px solid red">alabana.inane@mail.in</td>
    <!--<td style="border:1px solid red">France</td>-->
  </tr>
  <caption> European Customers</caption>
</table>
```




```
<h2>HTML Form</h2>
<form action="handleSubmit()">
  Your name:<br>
  <input type="text" name="firstname" value=""><br>
  <input type="radio" name="gender" value="male" checked> Male<br>
  <input type="radio" name="gender" value="female"> Female<br>
  Have you been to Asia or Europe?<br>
  <input type="checkbox" name="travel1" value="Asia"> Yes, I have been to Asia<br>
  <input type="checkbox" name="travel2" value="Europe"> Yes, I have been to Europe<br>
   Your favorite color:
  <input type="color" name="favcolor"><br><br>
  Your Birthday:
  <input type="date" name="bday"><br>
  Range of children's age(if any):
  <input type="range" name="points" min="0" max="10"><br><br>
  Telephone:
  <input type="tel" name="phone" pattern="[0-9]{3}-[0-9]{2}-[0-9]{3}"><br><br>
  Select a week for appointment:
  <input type="week" name="year_week"><br>
  Select a time for appointment:
  <input type="time" name="usr_time"><br>
  Add your Website:
  <input type="url" name="homepage"><br>
  <input type="button" onclick="alert('I am a regular button!')" value="Click Me!">
  <input type="submit">
   <input type="reset">
</form>
```




```
<form action="handleSubmit()">
  Your name:<br>
  <input type="text" name="firstname" value=""><br>
  Your Education:<select name="education">
  <option value="classX">Tenth</option>
  <option value="Inter">Intermediate</option>
  <option value="graduate" selected>Graduate BE/Btech/BSc/BA/Bcom</option>
  <option value="postgrad">PostGraduate</option>
</select><br>
Comments:<textarea name="message" rows="4" cols="20"></textarea><br>
Experience:<textarea name="message" style="width:200px; height:100px;">
</textarea><br>
City of Preference:<input list="city">
  <datalist id="city">
    <option value="Hyderabad">
    <option value="Bangalore">
    <option value="Chennai">
    <option value="Pune">
    <option value="Mumbai">
  </datalist> <br>
 <input type="submit">
   <input type="reset">
</form>
```



CHAPTER 2

HTML - Creating the Web Content

“Content is king.” ~ Bill Gates

Having gone through the brief history of the web and JavaScript and how it has grown over the years, we will start our journey of modern web development right from the basics.

In this chapter, we will learn about the first and most important foundation pillar of web development—HTML (HYPERTEXT MARKUP LANGUAGE).

HTML is actually not a programming language, but it is a markup language. Markup languages are designed for the definition, presentation, and processing of text. It specifies the format of the code. The code is designed in such a way that it is syntactically distinguishable. HTML is the language of the web that web browsers use to compose text, audio, images, videos, graphics, and other materials into rich web pages.

Structure

- Getting started with HTML
- Building blocks of HTML
- Structure of an HTML document
- Different HTML elements

Objective

At the end of this chapter, you will be able to learn about the features of HTML and how to use HTML to put your content in the form of basic web pages.

Getting started with HTML

HTML code is plain text which can be saved with .html or .htm extension and opened with the browser. The following code is how a basic HTML code looks like:

```
<!DOCTYPE html>
<html>
<head>
<title>My First HTML</title>
</head>
<body>
<h1>Hello World!</h1>
<h2>Welcome to learning HTML!</h2>
<p>Welcome to Web Development!</p>
```

```
</body>
</html>
```

You can write the preceding code snippet in a simple notepad, but we will eventually need a specialized code editor as we progress in our web development, so now is the time to select an editor.

There are many editors available and you can choose whichever you are comfortable with. Here is a popular list to choose from:

- Visual Studio Code (**https://code.visualstudio.com/download**)
- Atom (https://atom.io/)
- Notepad++ (**https://notepad-plus-plus.org/download**)
- Sublime (**https://www.sublimetext.com/download**)

In this book, we will be using Visual Studio Code.

So, let's get started by downloading the Visual Studio Code (or whichever editor you want to use) from the link provided.

Now, create a new file, write the HTML content, and save as helloWorld.html.

Next, open the HTML file with the Chrome browser (or any other browser) to see the output.

This is how the code looks in the editor.

Figure 2.1: *helloWorld.html*

And this is how the browser output looks like:

Figure 2.2: *Browser output for helloWorld.html*

For starters, the code does look weird with all the < > but the part of the code within < > is not seen in the final browser output. So, what are these and where did they disappear?

The final output renders the text in different sizes, which we did not specify.

How did this happen?

Let's dive into HTML to get an answer to these questions and much more!

Building blocks of HTML

The HTML content comprises three building blocks, which are as follows:

- **HTML elements:** HTML elements are the building blocks of any HTML web page. An HTML element comprises a pair of tags (`<starting-tag>` and `</closing-tag>`) and the content which goes between the tags.

HTML elements can be nested within other HTML elements. This results in a tree-like structure where the nodes have a parent-child relationship due to the nesting.

- **HTML Tags:** An HTML tag surrounds the content to be displayed and applies a special meaning to it. The tags are written between the < and > brackets and do not appear in the final output. They add extra meaning and associate attributes to the content which they surround. The browser reads an HTML document from top to bottom and left to right and renders the content.

  Here are some points to keep in mind regarding HTML tags:

- `<tag> CONTENT </tag>`: This tag represents the name of the tag; for example, `html`, `head`, `body`, `div`, and so on.
- All tags follow this format except the empty tags which have no content. Each tag has some special meaning which directs the browser to handle its content in a different way.
- Empty tags follow the format `<starting-tag>` or `<starting-tag/>`. They do not need an explicit closing tag as they have no content.
  - `<br>`: This is an empty tag which introduces a line break.
  - `<hr>`: This is another empty tag which draws a horizontal line on the screen.
- **HTML attributes:** HTML attributes are special properties attached to the HTML elements to impact their behavior. The attributes follow the following syntax:

```
<tag attribute-name="attribute-value" > CONTENT </
tag>
<p style="color:blue;"> Styled Paragraph Content </
p>
```

`style` is an attribute which can be attached to almost all the tags to apply additional styling. This is an inline style which gets directly applied to the element. We will learn more about styling using CSS in *Chapter 3, CSS – Making the Content Beautiful*.

Structure of an HTML document

An HTML document comprises a tree-like structure. The first level is the HTML tag, the root element of the page which comprises the two main elements: head and body.

The head tag is the optional part syntactically as it defines the details about the page like the title, the metadata information, link, style, script, and base.

The body is the part which contains the actual content which gets rendered on the page. The body can contain any number of elements which will all get displayed.

The HTML web page structure can be depicted as follows:

Figure 2.3: *HTML page structure*

The HTML page is like a tree structure with each element forming a node of the tree and different nodes having a parent-child relationship between them starting from the root node (HTML). This tree can be depicted as follows:

Figure 2.4: *HTML code and the corresponding DOM element tree*

There are many containing and grouping tags which make up the complete body of the HTML depending on the usage and the requirement. The container tags can be divided into the following two main categories on the basis of their default behavior of how they get rendered on the page as follows:

- **BLOCK-LEVEL elements:** These are the main container elements which render the content in a new line. They have the following main features:
  - They start with a new line.
  - They take up the complete width available.
  - For example, `div, p, h1-h6.`
  - In our first example, we saw that each of the `p, h1, h2` started on a new line even though we did not give any explicit line break.
- **INLINE elements:** These are the container components used to render content on the same line as the preceding content. They have the following features:
  - They do not start with a new line of their own.
  - They take up only as much width as is needed by its content.
  - For example, `span`.
  - The following example shows the use of both block and inline elements:

Figure 2.5: *HTML code using block-level and inline elements and the corresponding browser output*

HTML elements

In the following sections, you will learn about the different HTML

elements, how and where they can be used to define content, and the various attributes attached to them.

Basic HTML tags to define content

In this section, we will go through the basic content-related HTML tags which can be used for rendering content on the page by defining sections or adding special meaning and/or special formatting to the content. One common feature regarding these tags is that they do not have any mandatory attributes attached to them. They can be styled using the optional style attribute but do not have any attribute which defines their specific intent and behavior on the page.

The following table lists out all the basic HTML tags with their purpose and basic usage:

HTML Tag	

<html>	The root tag which contains everything else on the page.
	`<html>`
	`<!--Everything else comes here-->`
	`</html>`

<head>	This tag contains the title of the page and links tags to include external stylesheets, external fonts, and metadata regarding the content of the web page. These details do not show up on the page output.
	`<title> My First Page</title>`
	`<link rel="stylesheet" href="styles.css"/>`
	`</head>`
	`</html>`

<body>	This tag contains the nested tree of elements which forms the complete content which gets rendered on the web page.
	`<head>….</head>`
	`<body>`
	`<div> This is what shows up on the page</div>`
	`</body>`
	`</html>`

** **	This is the block-level container element which is used to group together and render content in the page.
	`<head>….</head>`
	`<body>`
	`<div> This is what shows up on the page<div> There can be more nested content </div></div>`
	`</body>`
	`</html>`

<h1>, <h2>, <h3>, <h4>, <h5>, <h6>	HTML provides a set of heading tags which are different in sizes and can be used for different levels of headings on your page. These hold importance as per the levels and are used by search engines to index the content of the page based on the headings. By default, there is a size associated with each of the headings which can be changed using CSS (which we will explore later).
	`<h1>Level 1 Heading</h1>`
	`<h2>Level 2 Heading</h2>`
	`<h3>Level 3 Heading</h3>`
	`<h4>Level 4 Heading</h4>`
	`<h5>Level 5 Heading</h5>`

```
<h6>Level 6 Heading</h6>
</body>
</html>
```

Paragraph tag: The `<p>` tag is used to render any plain text in the form of a paragraph. Any extra spaces and line breaks are removed in the final content which gets rendered.
If you want to include a new line, you can make use of the **`<br/>`** empty tag.

```
<body>
<h1>…..</h6>
<p> This is what shows up on the page
No spaces are retained
No new lines are retained
The final output is a plain paragraph<br/> This will introduce a
line break</p>
</body>
</html>
```

Preformatted text: If you want to retain the spaces and lines in the content as it is, you cannot make use of the **`<p>`** paragraph tag. For this purpose, the `<pre>` tag is useful as it will render the content with all the spaces and new lines as given.

```
<title> My First Page</title>
<link rel=”stylesheet” src=”styles.css”/></head>
<body>
<h1>….</h6>
<p> …..</p>
<pre> Web development is so much fun,
There are many more to come!!
Understand each TAG
And there will be no DRAG!!
</pre>
</body>
</html>
```

Adding tags applying special formatting: There are many tags which apply special formatting to the content inside them.

- `<b>`, `<strong>`: This tag makes the content in bold format. **`<strong>`, `<sub>`,`<sup>`, `<ins>`, `<del>`, `<mark>`**
- `<em>`: This tag renders the content in italics but has some extra emphasis attached to the content.
- `<small>`: This tag renders smaller size content.
- `<sub>`: This tag renders content as subscript.
- `<sup>`: This tag renders content as superscript.
- `<ins>`: This tag renders content as edited and added.
- `<del>`: This tag renders content as striked and deleted.
- `<mark>`: This tag renders content as highlighted.

Table 2.1: *Basic HTML content tags*

Now, we will see our example with all the preceding tags put to use and see how they appear on the browser:

Figure 2.6: *Example with basic tags*

Special content-related tags

Now, if we have to add some differently formatted data like tables, bulleted/numbered lists, hyperlinks, images, video, audio, we will see how we can include these in our web page.

HTML hyperlink

The anchor `<a>` tag is used to render hyperlinks in the web page. The attributes to be used with the anchor tag include the following:

	Mandatory (Y/N)	Default Value if N
href	Yes	None
This attribute is used to provide the URL to which the link will direct to. It can be an external website address, a page in the same application, or a bookmark on the same page.		
target	No	_self
This attribute indicates the location where the link will open up. The possible values for this attribute include: ● `_blank`: The link opens in a new window or tab. ● `_self`: The link will open in the same place from where it was clicked. ● `_parent`: The link will open in the parent frame of the frame containing the link. ● `_top`: The link will open in the complete window. This will be different from _self if your link is within a child frame of your complete window, else it appears to be same as _self. ● `framename`: The link will open in a named frame within your complete window definition.		

Table 2.2: *Link-related attributes*

The following diagram shows the use of iframe and setting the target as the frame name:

Figure 2.7: *HTML hyperlinks*

HTML list

A list is a very important and popular content construct to show a series of content, related or unrelated. The list can be a numbered list, a bulleted list, or with no special indication.

The list is rendered using the combination of two types of tags: header level tag and list item level tags.

Header level tag:

The header level tag is used to bind together all the list items to appear on the list:

- `<ul>`: This is used as the header level tag for an unordered list which is indicated as a bulleted item list.
- `<ol>`: This tag is used as the header level of an ordered or numbered list.

List item level tag:

The list item level tags `<li>` are the same irrespective of the list type.

The following figure shows the ordered list code and browser view:

Figure 2.8: *HTML unordered list code and browser view*

Changing the header level tag to `<ol>` changes the look of the list item marker to a numbered list as follows:

Figure 2.9: *HTML ordered list code and browser view*

In *Figure 2.8* and *Figure 2.9*, we see the default list item marker for both unordered and ordered list.

Now, we will look into the styling properties and attributes related to the HTML list in the following table:

Mandatory/Optional	Default Value if N
This property is used to change the list item marker for an unordered list. The following values can be used: ● disc (default) ● circle ● square ● None The value is set using: `<ul style="list-style-type:square"></ul>` This property can be used to set the list item marker as an image. This attribute is used to set the list item marker numbering type for a numbered list. By default, it is '1', thus indicating a numbered list. Other values are as follows: ● `type="A"`: Alphabetical numbering ● `type="a"`: Lowercase alphabetical ● `type="I"`: Roman numbering ● `type="i"`: lowercase roman This attribute can be used in an ordered list to start the numbering of the list from a specific value.	

Table 2.3: *List related attributes*

The lists can be nested to any number of levels with the list items having another list within its contents and so on as shown in the following figure:

Figure 2.10: *Nested List with a combination of ordered and unordered lists*

Now, we know how to include any type of list in our content. Let's look at tables, another way of presenting data in a very clear and easy-to-understand format.

HTML table

Data presented in the form of a table appears clearer and is more comprehensible to the human brain like a picture. Instead of looking through loads of paragraphs of data, it is much easier to look for information in a table as the data is segregated and laid out categorically.

A table format comprises *m x n* (*m by n*) number of rows and columns, *m* rows, and *n* columns. The first row is generally the header row which contains the column headers to indicate what each column data consists of.

The HTML tags used for rendering a table also follow a similar pattern to render the two-dimensional data. The basic table-related HTML tags include the following:

Tag	Purpose
`<table>`	This tag is used to define the overall table layout and contains all the table data.
`<tr>`	This tag is used to define each table row.
`<th>`	This tag is used to indicate the table header cells inside the table row.
`<td>`	This tag is used for the table data cells which contain the table content.

Table 2.4: *HTML table main elements*

The basic table with basic table tags looks something like the following figure showing the code and the browser view:

Figure 2.11: *A simple table using the table tags without any extra styling used to render data*

But *Figure 2.11* obviously does not look like a proper table. Let's apply some minimal styling to our tables using attributes (we will have to include some CSS here to make a basic HTML table look like one). The properties and attributes which can be applied on the HTML table are listed in the following table:

Property/Attribute	Purpose
`border`	As the name suggests, this styling property is used to add borders and that's what we need for our tables. This is added to the table as well as to each cell data (table, tr and td).
`border-collapse`	This property is used if we want the adjoining borders to collapse into a single border line. This property has to be applied at the table level. The default value of the property is separate.
`padding`	This is a CSS property to add some padding space between elements and can be used to add extra padding space between the table cell data.
`border-spacing`	Instead of collapsing, if we want to have some space between the different borders, we can use this CSS property at the table level. It will have no impact for the collapsed border.
`rowspan="n"`	This attribute can be used to indicate if the content spans to more than one row. The content will get extended and occupy the number of rows given for this attribute.
`colspan="n"`	This is similar to rowspan but with respect to columns.

Table 2.5: *Table styling attributes and properties*

The **`caption`** tag can be included in the table definition to add a caption to the table.

Now, we will apply these attributes and properties and style our initial basic table and now the table will look like this:

Figure 2.12: *Example of styled HTML tables using the attributes and properties*

The example shows two tables in *Figure 2.12* which use the different values of border-collapse and other properties which can be applied on an HTML table.

So far, we have seen different ways to render the content on the screen in different formats. What if we want the user to be able to enter some data into the content? Like the contact us form, we see on almost every website.

Next, we will learn how to include forms with different types of elements to enable the user to enter data and interact with the website.

HTML form

A form provides the capability to the user to be able to enter data, give feedback, leave comments, select options, upload files, and interact with the web page in different ways. HTML provides the <form> element using which we can define a form in our page to be able to accept user inputs.

```
<form>Form elements …</form>
```

The form comprises form elements which are the actual elements on the page with which the user interacts.

Input form element

The most useful and common form of element which covers most of the form element types is the `<input>`. By giving different values to the type attribute, we can define different types of form elements as listed as follows:

Input type	Purpose
`button`	This adds a button to the form. The button handler can be added to `onClick` event.
`checkbox`	This type is used to include a checkbox element where the user can select multiple choices from the given options.
`color`	This adds a color picker using which the user can select a color.
`date`	This type adds a text box associated with a calendar enabling the user to select a date.
`datetime-local`	This type adds a text box associated with a calendar enabling the user to select a date and time input field, with no time zone.
`email`	This is a text box which expects the content in email format.
`file`	This is a browse file button that enables the user to select a file for upload.
`hidden`	This type is used to define a hidden field not visible in the UI. This may be useful for derived values which should not be shown to the user.
`image`	This defines an image as a submit button. The path to the image is specified in the src attribute.
`month`	This text box is associated with a calendar enabling the user to select a month and year.
`number`	This is a numeric field for which restrictions on range can be set using attributes like min, max.
`password`	This is a text box with masked data for entering password data.
`radio`	This is a radio button that can be added where the user can select only one of the given options.

This is a control to set a range with attributes min, max to set the start and end of range, and step attribute to set the interval.
This is a reset button on click of which all the values of the form elements are reset to initial values.
This is a normal text field which can be used as a search field.
This is a button which becomes the submit action handler for the complete form.
This is a text box which expects the content in telephone number format.
A single line text box is added using this.
This is a text box enabling the user to enter time (without timezone).
This is a text box which expects the content in the URL format.
This text box is associated with a calendar enabling the user to select a week and year.

Table 2.6: *Input tag with different type values*

Using input, we have been able to add so many different types of elements, just by using a different value of the type attribute as shown in the following figure:

Figure 2.13: *HTML basic form showing different input types*

Other form elements

Apart from the input form elements, there are some other form elements for the other type of constructs to be displayed on the form as listed in the following table:

Form element	Purpose
This element is used to add a drop-down list with a fixed number of options to select from. The option marked with the selected attribute will be selected by default.	`<option` `<option value=”graduate” selected>Graduate BE/Btech/BSc/BA/Bcom</option>` `<option` `</select> value=”postgrad”>PostGraduate</option>` `value=”Inter”>Intermediate</option>`
This element is used to include multiline text area to enter detailed comments, descriptions, and so on. The visible size can be set by giving value to the rows and cols attributes. Also, the style attribute can be used to set the width and height.	
This element is used to associate an input text with a list of predefined options. The list attribute of the input is set to the id attribute of the datalist element.	`<option value=”Hyderabad”>` `<option value=”Bangalore”>` `<option value=”Chennai”>` `<option value=”Pune”>` `<option value=”Mumbai”>` `</datalist>`

Table 2.7: *Other HTML form elements*

The following example shows the code and browser view using the other form elements:

Figure 2.14: *Other form elements*

Special meaning-related HTML5 tags

Some semantic tags were added as part of HTML5 which do not have any special styling attached to the content, but they associate some special meaning to the content which is used by SEO crawlers, machine readers, and other reading software. The following table lists out the main semantic-related HTML5 tags:

HTML element	Purpose
<article>	This tag defines an article in a document which is an independent part of the page and makes complete sense on its own.
<section>	This tag is used to divide the entire page into different parts wherein all parts may be interrelated and/or related to the subject of the web page.
<header>	This tag is used to define the header for the complete document or a specific section.
<footer>	This tag is used to define the footer for the complete document or a specific section.
<aside>	This tag is used to define some side content like sidebar which is related to the overall content.
<bdi>	This is used to render text that may be written in a different direction related to the other content outside the tag
<main>	This tag is used to render the main content of the overall web page.
<summary>	This tag renders a visible heading for a <details> element.
<details>	This is used to add extra details for the summary which the user may view or hide.
<dialog>	This is used for a dialog box or window.
<figcaption>	This is used to add a caption for a figure tag.
<figure>	This is used to render a self-contained content like an image, diagrams, and so on.
<mark>	This is used to highlight text in between content.
<meter>	This tag is used to define a scalar measurement within a known range (a gauge).
<nav>	This is used to define navigation links.
<progress>	This is used to indicate the progress of a task. For example, <progress value="40" max="100"></progress>
<wbr>	This tag is used to indicate a possible line-break if the content exceeds in the given container size.

Table 2.8: *HTML5 semantic tags*

HTML comment tags

Any programming is incomplete without comments and need for commenting code. HTML comments can be added to the HTML code as follows:

```
<!-- Whatever comes here are considered as comments and
not displayed on the browser -->
```

HTML images

Images are another integral part of any web content as the popular saying goes: ‘A picture is worth a thousand words.’ They can enhance the appearance of your website.

The `<img>` tag is used to include images in the body content. The `<img>` tag is by itself an empty tag and contains only attributes.

The main attributes for the `<img>` tag are as follows:

- `src`: The `src` attribute specifies the location (URL or web address or local address) of the image, from where the image will be used. This is mandatory.
  - This can be directly a web URL like:
    ```
    <img src="https://www.pexels.com/photo/person-walking-on-green-field-2138908/" alt="A person walking on green field>
    ```
  - This can be a folder path pointing to a specific location on your system:
    ```
    <img src="C:/Pictures/Chapter2/2.3.jpg" alt="HTML Web Page Structure">
    ```
  - This can be a relative path with reference to the same folder as the HTML file:
    ```
    <img src="/2.3.jpg" alt="HTML Web Page Structure">
    <img src="/html/2.4.jpg" alt="HTML Tree Structure">
    ```
- `alt`: The `alt` attribute provides an alternate description for the image, if the mage is not rendered due to any reason like unavailability of image, wrong URL, poor connection, and so on. If the image is not shown, this text will be shown. This text should give details of the image. This is not mandatory but it is a good practice to always provide this value.
  ```
  <img src="img_rainbow.jpg" alt="Rainbow showing up">
  ```
- The `width` and `height` attributes can be used to set the size of the final image.
- The `style` attribute can also be used to set the size of the image using the `width` and `height` properties.

HTML character entity references

HTML character entity references are a special set of characters in the form of a special code, which the browser understands and displays as another special character or a symbol associated with the entity reference code.

These generally follow the format of starting with &, with some code and followed by; and are case sensitive.

This is helpful in specially two cases:

- If we need to display a symbol, which is not readily available on the keyboard. For example, the copyright symbol ©; this can be

displayed by using the code ©

- If we need to display reserved characters as part of the text. For example, if we need to print < or > symbols as part of the content, it will break the HTML page as the browser will confuse it with some tag. So, we can make use of the entity reference code instead:

Entity Reference Code	Symbol
<	<
>	>
&	&

Table 2.9: *HTML commonly used entity reference codes*

You can check the exhaustive list at **https://www.w3schools.com/html/html_entities.asp**.

Conclusion

Being able to use HTML to build web content is the first step to web development. Now, we can add the basic content and build a basic website. Though it looks raw without any styling, but it is a mandatory aspect of any web page as content is the soul of any application. With the soul in place, we will now move on the styling the content with CSS and making it interactive with JavaScript. In this chapter, we created plain content with HTML. In the next chapter, we will style our content with CSS and make it more beautiful.

Questions

1. Which attribute/property is used to change the list item marker of an ordered list?

   A. list-style-type

   B. list-style-image

   C. type

   D. list-item

   Answer: Option C

   For an ordered list, the type property can be used to change the item marker. List-style-type and list-style-image are styling properties used on unordered lists.

2. The new semantic tag which defines a part of the document, which is independent and makes complete sense on its own.

   A. Section

B. article

C. Header

D. summary

Answer: Option B.

An article is a part in a document which is an independent part of the page and makes complete sense on its own.

3. Any extra spaces and line breaks removed in the final content which gets rendered in ___, whereas all content will retain spaces and special formatting in ____.

   A. `<div>, <p>`

   B. `<div>, <pre>`

   C. `<pre>, <p>`

   D. `<p>, <pre>`

   Answer: Option D.

   Any extra spaces and line breaks are removed in the final content which gets rendered are removed for p tag. And retained for a <pre> tag.

4. Which of these are the valid values for type attribute for input field in HTML forms?

   A. week

   B. date

   C. datetime-local

   D. All of the above

   Answer: Option D.

   All the values given, date, week, datetime-local, are valid values.

5. This tag is used to indicate a possible line break if the content exceeds in the given container size.

   A. mark

   B. em

   C. wbr

   D. break

   Answer: Option C.

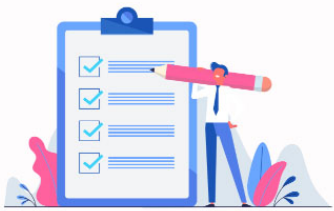
   The `<wbr>` tag is used to indice.

Organising always seems impossible until its DONE!

TODO

Login using

By creating an account you Agree to our Terms of Conditions



Organising always seems impossible until its DONE!

TODO

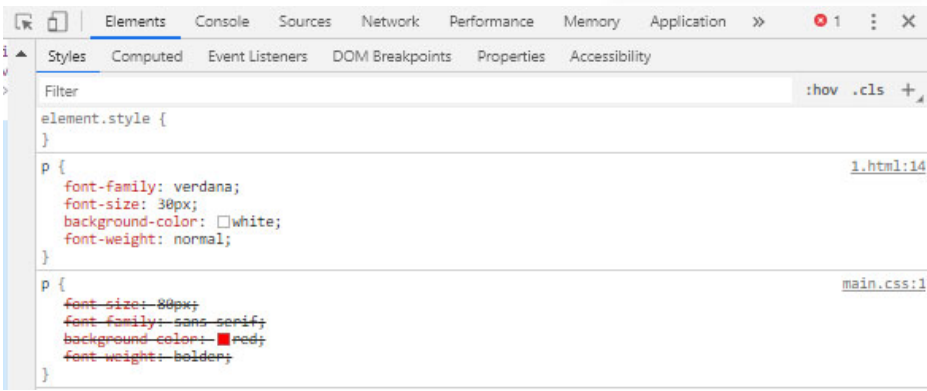
Login using

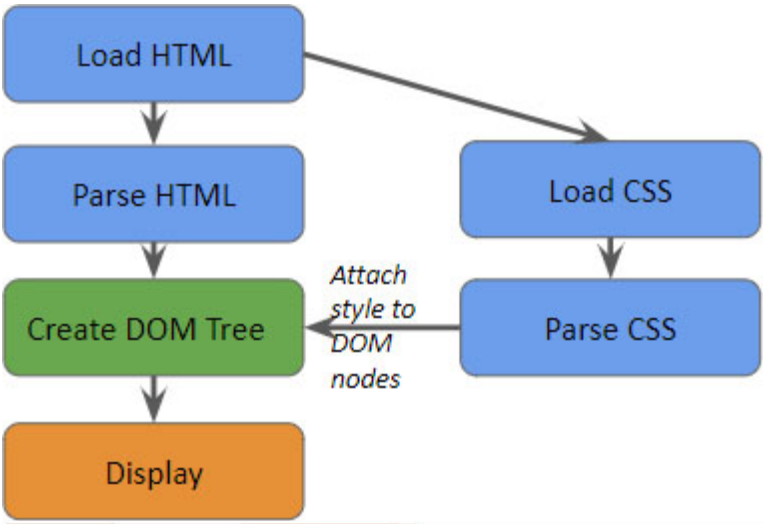
By creating an account you Agree to our Terms of Conditions

Normat text

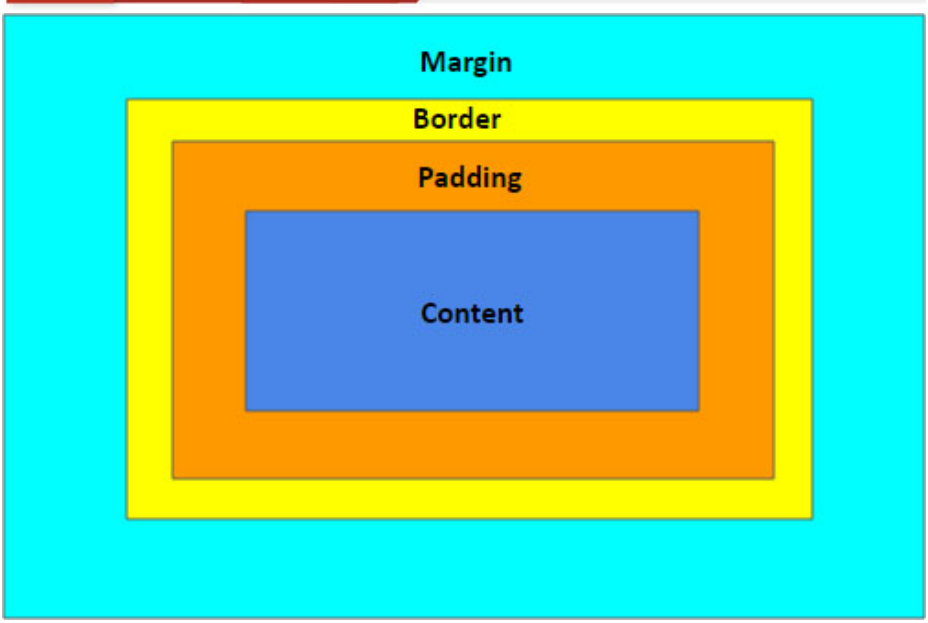
This is bold text Inline style text

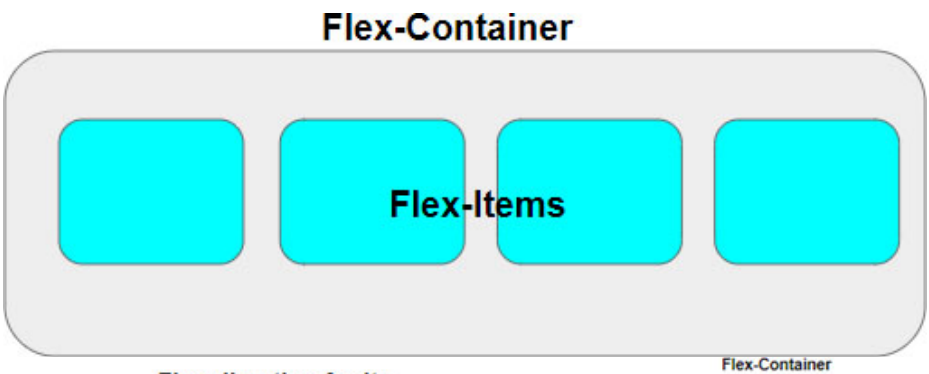
This is an external styled paragraph



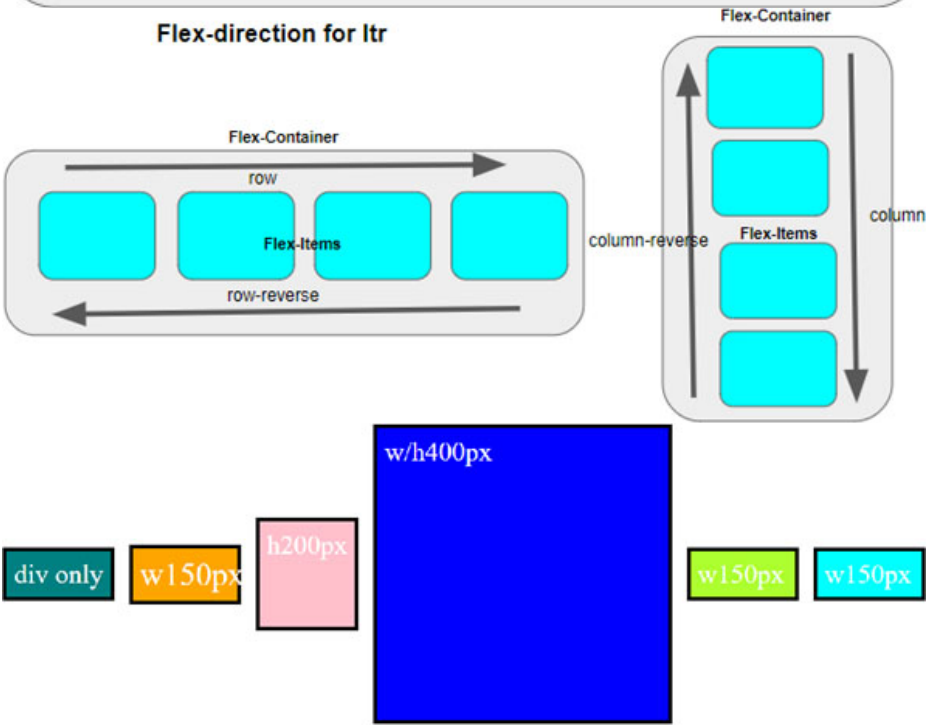


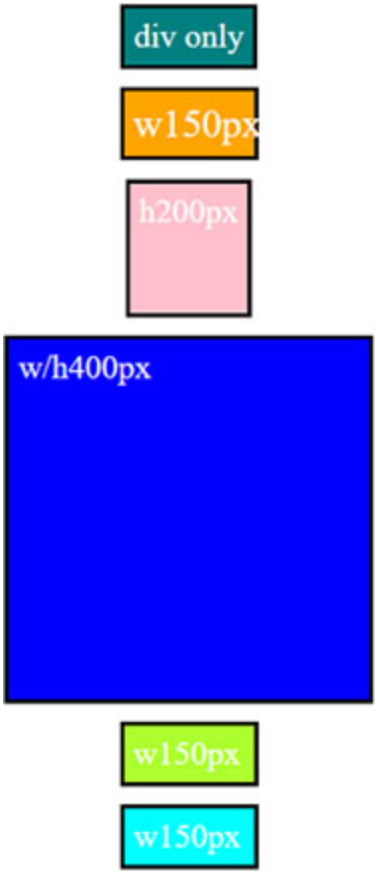
(space)	**Descendant selector**	• Matches all elements that are descendants of a specified element • element1 element2
>	**Child selector**	• Matches all elements that are the immediate children of a specified element • element1> element2
+	**Adjacent Sibling Selector**	• Matches all elements that are adjacent siblings of a specified element • element1 + element2
~	**General Sibling Selector**	• Matches all elements that are siblings of a specified element. • element1 ~ element2





Flex-direction for ltr





CHAPTER 3

CSS - Making the Content Beautiful

"UI is the saddle, stirrups, & the reins. UX is the feeling you get being able to ride the horse."

— Dain Miller, Web Developer

"Intuitive design is how we give the user new superpowers."

— Jared Spool, Web Site Usability: A Designer's Guide

In the last chapter, we learned how to add different types of content in different formats and having different semantics, into our web page using HTML. In this chapter, we will look into the second pillar of web

development. This is the pillar which adds all the beauty and poise to our content—**CSS (Cascading Style Sheets).** CSS takes complete control of styling the web and strengthens the content with visual design, the power of which is boundless. It improves the intuitiveness of the web, along with enhancing its appearance. CSS is an absolute must have skill in web development.

Structure

- Power of CSS3
- Basics of CSS
- CSS precedence and specificity
- CSS styling – Font, text, background properties
- CSS – The box model
- Positioning elements
- Responsive design

Objective

After studying this chapter, you will learn about the basic syntax and properties of CSS and how it can be used to our advantage to improve the overall appearance of the web pages.

Power of CSS

To understand and realize what CSS can do, you need to see this webpage in the following two forms.

The first diagram is of the webpage without any styling applied:

Figure 3.1: *A login page without CSS*

The next diagram shows the same content with CSS applied for styling:

Figure 3.2: *The same page after applying CSS*

What a drastic makeover!

That's the power of CSS!

You now know that HTML defines content and CSS defines the presentation of the content:

- With CSS, you can control the layout of multiple web pages all at once and make your website consistent
- With CSS, you can apply different styles to the same content to change the look for different media devices very easily

Let's begin by understanding the basic CSS code structure and what it comprises of.

Basics of CSS

CSS code structure is very simple to understand. The CSS code body comprises of CSS rule sets.

A CSS rule-set consists of two main parts, as follows:

- **A selector:** The selector points to the HTML element you want to style.
- **A declaration block:** The declaration block contains one or more declarations separated by semicolons. Each declaration/rule includes a CSS property name and a value, separated by a colon:
  - The property is the property to be modified
  - the value to be assigned to the property

Example:

```
p {
color: red;
 text-align: center;
}
```

Here `p` is the selector, and the remaining part with braces {} is the declaration block. Here the block has two rules where `color`, `text-align` are properties and `red`, `center` are property values which are being assigned.

CSS syntax

Points to keep in mind regarding the syntax:

- A CSS declaration always ends with a semicolon
- The property and value are separated by a colon
- Each declaration block is surrounded by curly braces
- By default, all CSS syntax is case-insensitive but with some exceptions:
  - Identifiers (including element names, classes, and IDs in selectors) are case-sensitive
  - Property values like URL, content are case-sensitive
  - Element names are case-insensitive for HTML DOCTYPE
  - Property values like color, font, and many more are not case-sensitive

That's all there is to the basic CSS Syntax!

How to include CSS in HTML?

Now that you know the basic syntax to write CSS, the next step is to include it in your HTML content so that it gets applied to the content. There are three ways of applying a CSS stylesheet to HTML:

- Inline style
- Internal style sheet
- External style sheet

Inline style

An inline style can be used to apply a unique style to a single element only. The `style` attribute is included in the relevant elements with the required properties to use inline styles.

The following example shows how to change the `color` and the top margin of an element:

```
<p style="color:red;margin-top:30px;">This is a
heading</p>
```

Internal style sheet

An internal style sheet can be used if one single page has a unique style. The styles defined using internal style sheet apply to the elements on the current page only. Internal styles are defined within the `<style>` element, inside the `<head>` section of an HTML page:

```
<head>
<style>
body {
 background-color: blue;
}
h1 {
color: maroon;
 margin-left: 40px;
}
p {
color: red;
}
</style>
</head>
```

External style sheets

This is the most flexible approach as the styles are defined in a separate file using.css extension. The same CSS file can be applied to any page which needs to use the same styling. Each page can include a reference to the external style sheet file inside the element. The element goes inside the head section of the HTML page as a link tag. The styling of a page can be changed just by changing the external CSS file:

```
<head>
<link rel="stylesheet" type="text/css"
href="mystyle.css">
</head>
```

Cascading styles

Can you guess why the styles are called cascading style sheets? Yes, they have a cascading effect on the way they get applied.

If multiple styles are applied on a page, it will be applied in a particular order of priority. Generally speaking, we can also say that all the styles will **cascade** into a new virtual style sheet by the following rules, where the first one has the highest priority:

- Inline style (inside an HTML element)
- Internal style sheets (in the head section)/External style sheet—Depending on the order in the head section, the one which comes later takes priority as it overrides the other one
- Browser default

The following code shows the inclusion of CSS using all the above three approaches:

```
<html lang="en">
<head>
<meta charset="UTF-8">
<meta http-equiv="X-UA-Compatible" content="ie=edge">
<title>uHost</title>
<link rel="shortcut icon" href="favicon.png">
<link rel="stylesheet" href="main.css">
<style>
 div {
  font-size: 40px;
 background-color: red;
  font-weight: bold;
 }
p{
    font-family: verdana;
     font-size: 30px;
```

```
    background-color: white;
    font-weight: normal;
 }
</style>
</head>
<body>
Normat text <div>This is bold text
<span style="font-family: verdana; font-size:30px;
background-color: blue;font-weight: normal;">
 Inline style text</span>
</div>
<p>
This is an external styled paragraph
</p>
</body>
</html>
```

The external style sheet included in the preceding HTML is as follows:

```
External style sheet – main.css:
p {
 font-size: 80px;
 font-family: sans-serif;
 background-color: red;
 font-weight: bolder;
}
```

The output for the above styling looks like below:

Figure 3.2.1: *Output with styles applied*

The last paragraph's external style got cascaded by the internal style. Hence the background is white. You can see that if you open the elements tab in your chrome debugger as below:

Figure 3.2.2: *View of the debugger showing the cascading styles*

Working of CSS

When a browser displays a document, it has to combine the document's HTML content with its CSS style information. It processes the document in two stages in the following steps:

- The browser loads and parses the HTML and CSS and converts into the **DOM (Document Object Model).**
- The DOM represents the document in the computer's memory.
- Every Cascading Style Sheet is a series of instructions called statements. A statement does two things:

- It identifies the elements in an HTML document that it affects
- It tells the browser how to draw these elements

- The browser attaches the style to the associated DOM nodes and finally displays the contents of the DOM.

The following diagram shows the flow in which the HTML and CSS are loaded and parsed to create the DOM tree which gets displayed finally:

Figure 3.3: *How browser works with CSS and HTML?*

Let's now look into each of the parts of CSS syntax in detail and how we can write CSS rules.

Properties

A property is assigned to a selector in order to manipulate some aspects of its style. Examples of properties include `color`, `margin`, and `font`.

Values

The declaration value is an assignment that a property receives. It is the value assigned to the property, so it is tightly coupled to the property. For example, the property color could be assigned the value red or blue.

Some basic value sets assigned to properties used in CSS include:

- **Pre-defined options/values:** For example, `display: none; position: absolute;`
- **Colors:** Colors can be given in different formats, the color names, Hex codes, RGB values. For example, `background-color: red; font-color: #FFFFFF;`
  - **Length, sizes and numbers:** For example, `width:50%; height: 200px; z-index:100;`
  - **Functions:** For example, `transform: scale(10,10,10);`

Selectors

Selectors are patterns which are used to select the HTML elements for styling. This is what represents the part of the web page to which the styles will get applied.

Universal selector:

The universal selector represented by an asterisk (*) is used when we need to apply a style to all elements. Optionally, it may be restricted to a specific namespace or to all namespaces.

Syntax: * ns|*

Example: * will match all the elements of the document.

This is not recommended to be used as it is inefficient in the way it parses all elements.

Instead, it is always better to use a `<body>` element selector as it explicitly selects everything inside the body of the HTML page.

Element or Type Selector:

The basic selector is the HTML element. This is used to select all elements that match the given node name. Any HTML element can be a possible CSS selector. The selector is simply the element to which the styling is applied. For example, the selector in the following code is p (paragraph element):

```
p { text-indent: 3em }
```

Class selectors:

This selector is used to group different elements and apply the same style. A simple selector can have different classes, thus allowing the same element to have different styles. For example, an author may wish to display p in a different color depending on its position on the page:

```
p.header { color: #191970 }
p.textcss { color: #4b0082 }
```

The preceding example has created two classes, `header`, and `textcss` for use with HTML's p element. The classattribute is used in HTML to indicate the class of an element. For example:

```
<p class="header">This is header </p>
<p class="textcss"> This is the body text</p>
```

Only one class is allowed while defining a selector. Multiple classes can be assigned to an element. Classes may also be declared without an associated element:

```
.note { font-size: small }
```

In this case, the note class may be used with any element.

A good practice is to name classes according to their function rather than their appearance.

The note class in the preceding example could have been named small, but this name would become meaningless if the author decided to change the style of the class so that it no longer had a small font size.

ID selectors:

ID selectors are individually assigned for the purpose of defining on a per-

element basis. It gets applied to a specific element uniquely. There should be only one element with a given ID in a document. This selector type should only be used sparingly due to its inherent limitations.

An ID selector is assigned by using the indicator # to precede a name.

For example, an ID selector could be assigned as such:

```
#myid{ text-indent: 20px }
```

This would be referenced in HTML by the ID attribute:

```
<p id="myid">Text indent 20px</p>
```

Attribute selector:

Attribute selector matches elements based on the presence or value of a given attribute. It can be used in the following ways:

Syntax	Purpose
[attr]	Used to select elements with an attribute name of attr.
[attr=value]	Used to select elements with an attribute name of attr whose value is exactly the same as the given value.
[attr~=value]	Used to select elements with an attribute name of attr whose value is a whitespace-separated list of words, one of which is exactly value, that is, containing a specified word.
[attr|=value]	Used to select elements with an attribute name of attr whose value can be exactly the same as the value or can begin with value immediately followed by a hyphen -.
[attr^=value]	Used to select elements with an attribute name of attr whose value is prefixed by the given value.
[attr$=value]	Used to select elements with an attribute name of attr whose value is suffixed by the given value.
[attr*=value]	Used to select elements with an attribute name of attr whose value contains at least one occurrence of the given value within the string.
[attr operator value i]	This is used to indicate if the above operators perform a case sensitive search or not. By adding i (or I) before the closing bracket causes the value to be compared case-insensitively (for characters within the ASCII range).

```
a[title] {
color: purple; }

[href="https://example.org"] {
color: green; }

[attr~="green"] {
color: green; }

[attr|="green"] {
color: green; }

[attr^="WTT"] {
color: red; }

[attr$="LTD"] {
color: red; }

[attr*="val"] {
color: red; }

[attr="value" i] {
color: red; }
```

Table 3.1: *CSS Attribute Selectors*

Combinators

A CSS selector can contain more than one simple selector. Between the simple selectors, we can include a combinator to indicate the relationship between the selectors. A combinator enables us to specify more than one

selector to be more precise on what we want to select.

Group of selectors: This can be used when you want to apply the same style to several elements, so instead of creating different blocks for each one, you can simply list them using comma separator(,) and write the rules once:

```
h1, h2, h3, h4, h5, h6 {
 margin: 1rem;
 text-decoration: underline;
}
```

The preceding rules apply margin and a `text-decoration` to all heading tags at once.

The following diagram summarizes the combinators in CSS:

Figure 3.4: *CSS combinators*

Contextual selectors/Descendant combinators

Contextual or descendant selectors are strings of two or more simple selectors separated by white space. The descendant selector matches all elements that are descendants of a specified element.

These selectors can be assigned normal properties, and due to the rules of cascading order, they will take precedence over simple selectors.

For example, the contextual selector inP EM {`background: yellow`}is P EM. This rule says that emphasized text within a paragraph should have a yellow background; emphasized text in a heading would be unaffected.

- `element1element2`: Selects all `element2` inside `element1`
- `div p`: Selects all `p` elements inside `div` elements

Child combinators/selector

The child selector selects all elements that are the immediate children of a specified element. The child combinator (>) is placed between two CSS selectors. It matches only those elements matched by the second selector that are the children of elements matched by the first

It is stricter than a descendent selector where the element2 can be the child of element1 at any level of DOM.

- `element1>element2`: Selects all `element2` immediately under `element1` in DOM:
  `div > p { background-color: yellow; }`

Adjacent sibling selector

The adjacent sibling selector selects all elements that are adjacent siblings of a specified element.

Sibling elements must have the same parent element, and *adjacent* means *immediately following*:

- `element1 + element 2: element2` must be immediately after element1 in DOM under same parent:
  `div + p { background-color: yellow; }`

General sibling selector

The general sibling selector selects~. This means that the second element follows the first (though not necessarily immediately), and both share the same parent:

- `element1 ~ element2: element2 follows element1 under same parent in DOM`
  `div ~ p { background-color: yellow; }`

Styling using Pseudo class

Pseudo-classes are used to match content in a more extensive way since they can refer to elements not directly represented in the DOM. They refer to the state of the element and are used to style a special state of the element.

They can be dynamically accessed based on user interaction. In order to use a pseudo-class, you have to add a : to the beginning of the class name:

Pseudo class	Usage
:focus	This will match the elements which are currently in focus, for example, when a user clicks to edit an input or selects a drop-down value.
:active	This will match the moment at the point when the user clicks an element. The style will get applied as long as the user holds the click over the element.
:hover	This will match the elements at the point when the mouse is currently over it.
:empty	This will match the elements that do not have any child elements under it.
:valid	This will match inputs that hold legal or valid values according to each input type. For example, an email or number type would be valid; it holds data in the expected format.
:checked	This will match for checkbox inputs only that are currently checked.
:link	This will match the anchor tags, which are unvisited links.
:in-range	This will match the ranged input elements that hold values within a specific range.
:not(selector)	This will match every element that is not the one passed as a parameter. This will reverse the selectorcondition.
:first-child	This will match the first child inside a specific parent. For example, div:first-child will match the first div inside of its parent.
:first-of-type	This will match the first element of a given type.
:lang(language)	This will match the elements that have an attribute of language with a specific value.
:nth-child()	This will match the element that is the nth element inside of a specific parent.
:nth-last-child()	This will match the element, which is the nth-child starting from the end.
:nth-of-type()	This will match the element that is the nth element of a given type inside a specific

parent.	
This is the same as nth-of-type but counting from the end.	
This will match the element that is the only element of its type inside a specific parent. If there is more than one of the types, it will not match anything.	
This will match the element that is the only child of a specific parent. If there is more than one child within that parent, nothing will be matched.	
This will match the input elements with no required attribute.	
This will match elements with a read-only attribute.	
This will match elements with no read-only attribute.	
This will match elements with a required attribute.	
This will match the document's root element.	

Table 3.2: *CSS styling using pseudo-class*

Styling with Pseudo-elements

A CSS pseudo-element is used to style specified parts of an element. Pseudo-elements can be used to refer to content that does not exist in the source code, but the content is generated afterward.

For example, first-line can be used to change the font of the first line of a paragraph. Check the following example:

```
selector::pseudo-element {
 property: value;
}
```

As a rule, double colons `(::)` should be used instead of a single colon `(:)`. This distinguishes pseudo-classes from pseudo-elements.

CSS inheritance

HTML elements can inherit CSS styles from their parent elements. This is called CSS inheritance. Not all styles are inherited from a parent or ancestor element. Generally, styles that apply to text are inherited, whereas borders, margins and paddings and similar styles are not inherited.

HTML elements can inherit styles from more remote ancestors too, and not just from their direct parents. As in this example:

```
<body>
<div>
<p>
       This text inherits the font-size of the parent
       div element.
</p>
</div>
<style>
     body {
```

```
        font-family: Calibri;
      }
</style>
</body>
```

In this example the CSS property font-family is set on the body element. This CSS property is then inherited by the `div` and `p` elements.

Now that we know the different ways we can define CSS rule sets, we will learn about another important aspect of how the rules get applied - Specificity.

CSS precedence and specificity

When multiple CSS rules are applied on the same HTML elements, and those CSS rules set some of the same CSS properties, finally, which styles end up being applied to the HTML element, is determined by CSS precedence.

In the following example, the property font-size is being applied to multiple elements:

```
<head>
<style>
      body {
        font-family: Arial;
        font-size: 14px;
      }
      p {
        font-size: 16px;
      }
.logo {
        font-family: Helvetica;
        font-size :20px;
      }
</style>
</head>
<body>
<div id="header">
<span class="logo">Super Company</span>
</div>
<div id="body">
<p>
      This is the body of my page
</p>
</div>
</body>
```

Let's understand how the precedence gets applied here.

CSS precedence rules

When the browser needs to resolve what styles to apply to a given HTML element, it uses a set of CSS precedence rules.

Given these rules, the browser can determine what styles to apply. The rules are:

- Use of !important after CSS properties
- Specificity of CSS rule selectors
- Sequence of declaration

Note that CSS precedence happens at CSS property level. Thus, if two CSS rules target the same HTML element, and the first CSS rule takes precedence over the second, then all CSS properties specified in the first CSS rule takes precedence over the CSS properties declared in the second rule. However, if the second CSS rule contains CSS properties that are not specified in the first CSS rule, then these are still applied. The CSS rules are combined—not overriding each other.

Use of !important after CSS properties

The `!important` instruction has the highest precedence of all precedence factors, which should be used carefully:

- If you need a certain CSS property to take precedence over all other CSS rules setting the same CSS property for the same HTML elements, you can add the instruction `!important` after the CSS property when you declare it.
- It is not a good practice to use `!important` as it overrides the normal CSS rules flow and specificity. Try to avoid the use of `!important`.
- Try to make better use of CSS cascading properties and increasing specificity by using more specific rules:

```
<style>
 div {
  font-family: Arial;
  font-size: 16px !important;
 }
.specialText {
  font-size: 18px;
 }
</style>
<div class="specialText">
```

```
 This is a special text.
</div>
```

Scenarios when you can use `!important`:

- When there is a global CSS file and inline styles are being applied by you or others. In this case, you could set certain styles in your global CSS file as important, thus overriding inline styles set directly on elements.
- If there is a requirement to override the rule with the highest specificity (for example, ID selector), !important can be used.

Specificity of CSS rule selectors

The specificity of a CSS rule depends on its selector.

If the CSS selector is more specific, the CSS property declarations will get greater precedence inside the CSS rule for the selector. The more specifically or uniquely, a CSS selector points to an HTML element, the higher will be its specificity. The specificity is determined by the type of CSS selector.

Here is a list of CSS selector specificity:

- **Inherited styles:** Lowest specificity of all selectors, since inherited style targets the element's parent, and not the HTML element itself
- **Universal selector *:** Lowest specificity of all directly targeted selectors
- **Element selector:** Higher specificity than universal selector and inherited styles
- **Attribute selector:** Higher specificity than element selector
- **Class selector:** Higher specificity than an attribute, element, and universal selectors
- **ID selector:** Higher specificity than a class selector
- **Inline style using style attribute:** Stronger specificity than ID selector
- Universal selector (*), combinators (+, >, ~, ' '), and negation pseudo-class (:not) have no effect on specificity.

The following code shows how the same rule can be applied using different selectors:

```
<body>
<style>
      body { font-size: 10px; }
      div { font-size: 11px; }
```

```
      [myattr] { font-size: 12px; }
.aText { font-size: 13px; }
      #myId { font-size: 14px; }
</style>
<div > Text 1 </div>
<div myattr> Text 2 </div>
<div myattr class="aText" > Text 3 </div>
<div myattr class="aText" id="myId" > Text 4 </div>
</body>
```

When you start combining and applying multiple CSS selector types in your CSS rules, it will become necessary to check the precedence of those selectors more carefully. Also, you will have to ensure that they have no overlap in targeted elements in order to achieve the desired styling, or else your styles may get overwritten.

How to calculate specificity?

A selector's specificity is calculated as follows:

- Count the number of ID attributes in the selector (= `a`)
- Count the number of other attributes and pseudo-classes in the selector (= `b`)
- Count the number of element or pseudo-element names in the selector (= `c`)

On concatenating the above three numbers, `a-b-c` gives the specificity for the selector.

The selector with the highest specificity gets applied.

Sequence of declaration

Rules that appear later in the CSS code override earlier rules if both have the same specificity. So finally, it is cascading over!

CSS units

Some of the CSS properties need units of measurement, so in order to be able to use and apply the values correctly, we should know the units which can be used. The main CSS units can be categorized as follows.

Absolute lengths

- Absolute values are applied without any relation to user settings. For example, `px`, `cm`, `mm`, `inches`, and so on
- If browser default settings are changed, it does not impact the

absolute px size of fonts

- `px` is most commonly used for specifying the absolute length in web development

Viewport lengths

Adjust the size of the element we apply to according to the viewport. The viewport lengths can be applied with the h and w. The viewport is the area where the browser renders the content. This is your screen minus the reserved space of the browser chrome. This is a responsive unit as it is dependent on viewport size and can also be applied to `font-size` to make it responsive.

Units include `vw, vh, vmin` and `vmax`:

- A value of `1vh` is equal to 1% of the viewport height.
- A value of `1vw` is equal to 1% of the viewport width.
- `1vmin` will be equal to 1% of the viewport height or width, whichever is smaller.
- `1vmax` will be equal to 1% of the viewport height or width, whichever is greater.

Font-relative lengths

Adjusts to the default font size in the browser settings:

- Em and Rem units are used for font size.
- Both `em` and `rem` are flexible, scalable units which are translated by the browser into pixel values, depending on the font size settings in your design.
- Em is related to the parent's font size, which is inherited and used to make the calculation of the current element size.
- Rem(Root em) is related to the font size of the root, i.e., by default, the browser font settings. If font-size set at the html element level, it will be taken as the base else it will be the font-size from browser settings(for example, 1 rem = 16px for Medium, 1 rem = 20px for Large font-size)

Percentage

Percentage is a special unit that gets applied to any element. Percentage unit is dependent on the position property of the element which determines the containing block of the element:

- **Position: fixed:** Containing block is Viewport

- **Position: absolute:** Containing block is Ancestor content + Padding
- **Position: static or relative:** Containing block is Ancestor content
- Percentage applied to font size takes the font size from browser settings as a reference to calculate the %

CSS styling – Font, text, background properties

Let's look into styling the different types of elements you will add to your HTML page. We will look into all the common aspects and the main properties of styling each of them.

Styling font

Setting and styling the font is an important aspect of displaying the text content. The font related properties are inherited and should be set at the body level to avoid duplicate styling at different levels.

You can search and include new fonts from Google Fonts. On clicking + for the selected icon, you will get the link to be included in your header to be able to use the font on your page. The following table lists the properties which can be used for selecting font-families and styling fonts:

Property	Example
This is used to set up the font family for the text to be rendered. A list of font families should be provided with the generic family as the last fallback option. Generic families include – serif, sans-serif, monospace.	`p{ font-family: "Times New Roman", Times, serif; }`
This is used to set the style of the font with values normal, italic and oblique.	`p{ font-style: italic; }`
This is used to set the size of font.	`p{ font-size: 20px; }`
This is used to set the weight like normal, bold.	`p{ font-weight: bold; }`
This is set the font as normal or small-caps.	`p{ font-variant: small-caps; }`

Table 3.3: *Styling font*

Styling text

There are some specific aspects with respect to which we will style the text being rendered on the screen as follows:

Property	Example
The `color` property is used to set the color of the text. The color can be provided in any format like regular name, hex code, rgb notation like RGB(0,0,0).	`h1{ color: green; }`
This property is used to set the horizontal alignment of the text. Values: `left,`	`p{ text-align: center; }`

```
right, center, justified.
This property is used for adding or removing decoration by using values like underline,
overline, line-through, or none. This can be used to remove default decoration
like underline of anchor tags by using none.
}
p {
This is used to change the text to lowercase, uppercase, or capitalize to change the first
letter to capitals.
  text-transform: uppercase;
}
p {
This gives the indentation of the first line of the content. Specially, useful for
indentation in paragraphs.
}
p {
This is used to specify the space between the lines.
  line-height: 0.8;
}
This specifies the space between words in the text.
  word-spacing: 10px;
}
This specifies the space between letters or characters in the text.
  letter-spacing: 5px;
}
This property is used to change the direction of text.
  direction: rtl;
}
```

Table 3.4: *Styling text*

Styling background

The background related styling properties include the following:

Property	Example
This is used to set the color of the background.	`background-color: rgb(255, 255, 128);`
This is used to set the background's origin whether it should start from the border, inside the border, or inside the padding.	`background-origin: border-box;` `background-origin: content-box;`
This sets the initial position of the background. It can be set to directions like `top, bottom, left, right` or giving percentage or length values.	`background-position: center;` `background-position: 25% 75%;`

This sets how the background images are repeated in the given area.	`background-repeat: repeat-y;`
	`background-repeat: repea`
	`background-repeat: space;`
	`background-repeat: round;`
	`background-repeat: no-repeat;`

This sets the size of the background.	`background-size: contain;`
	`background-size: 50%;`
	`background-size: 3.2em;`
	`background-size: 12px;`
	`background-size: auto;`

This is used to set one or more background images.	`background-image: url("image2.gif");`
This is used to define how much the background (color or image) should extend within an element. Values include `border-box, padding-box, content-box`.	

All or some of the above listed properties can be given under single shorthand property
of background.

```
background-attachment;
```

Table 3.5: *Styling background*

You can go through *Chapter 2* to refer to the basic styling properties for lists and tables.

CSS – The box model

Each element is interpreted in a box model which comprises of the following:

- Content
- Padding
- Border
- Margin

Figure 3.5: *The CSS Box Model*

Box-model

Each HTML element rendered is considered to be a box. The box has four parts (or layers):

- **Margin:** The outermost part is the margin between this HTML element to other HTML elements.
- **Border:** The second part is the border. The border sits inside the margin and surrounds the padding and content of the HTML element.
- **Padding:** The third part is padding. The padding sits inside the border and surrounds the content of the HTML element.
- **Content:** The fourth part is the content. The content of an HTML element is whatever is displayed inside the HTML element. This is typically a combination of text, images, and other HTML elements.

Each of these layers can be set using styles.

Note that for non-replaced inline elements, the amount of space taken up (the contribution to the height of the line) is determined by the line-height property, even though the borders and padding are still displayed around the content.

Setting width and height

You can control the width and `height` of an HTML block element box using the width and height CSS properties.

You cannot control the width and height of an inline element using CSS. It

equals the size of the content of the inline element. The `width` and `height` properties can be set using either `px` unit or in %:

- Setting width to 100% takes up the entire width of the container. This is the default behavior of block elements.
- Setting height to 100% takes up the available height given by the parent of the container. If we have to set this to 100%, the height should be set at the parent level to some absolute value starting from `html->body` and all containing parent to finally impact the element. If the height is set as 1000 px for the body element, then the p element within the body can have a height set to 50%, which will get calculated based on the parent height.

  By using px, the `width` and `height` can be set to a specific value.

  By default, the `width` and `height` get applied to the content only:

```
#theDiv {
width :300px;
height :200px;
}
```

So the actual width of the rendered element is including padding, margin and border size as follows:

```
total width = width + margin-left + margin-right +
border-left-width + border-right-width + padding-left +
padding-right
total height = height + margin-top + margin-bottom +
border-top-width + border-bottom-width + padding-top +
padding-bottom
```

Box sizing

The above mentioned behavior of calculating `width` and `height` is the default box-sizing property called `content-box`.

You can change how the browsers calculate the size of the HTML element with the `box-sizing` CSS property. The box-sizing CSS property is new in CSS 3.0.

The box-sizing property can take these values:

- `content-box`
- `border-box`
- `inherit`

The `content-box` value is the default value. This makes the width and height CSS properties set the `width` and `height` of the content box

alone. The inherit value means that the HTML element inherits the value of this CSS property from its parent HTML element.

The `border-box` value makes the browser interpret the width and height CSS properties as the width and height of the border-box. The border-box is everything inside the borders of the HTML element, including the border itself.

This can be set to `border-box` to include the padding and border in the given width and height of the element.

The box-sizing border-box is convenient for development, to estimate the content size, including the padding and border.

Margin is not included, so do not specify margin if not needed.

To apply `box-sizing` border-box to all components, we will have to use the universal selector as applying to the body gets overridden due to display block property, in spite of inheriting the property.

Vertical margin collapse

Another typical behavior of CSS margins is that, when you have two boxes with vertical margins sitting right on top of each other, they will collapse. Instead of adding the margins together as you might expect, only the biggest one is displayed:

```
p {
 padding: 20px 0 20px10px;
 margin-top: 25px;
 margin-bottom: 50px;
}
```

Each paragraph should have 50 pixels on the bottom, and 25 pixels on the top. That's 75 pixels between our elements, right? Wrong! There's still only going to be `50px` between them because the smaller top margin collapses into the bigger bottom one.

The important part here is that only consecutive elements can collapse into each other.

A good option to avoid margin collapse is to stick to a bottom-only or top-only margin convention.

There, three base cases are described:

- Adjacent siblings who both have margins
- A parent who holds one or more child elements where the first and/or last (or the only) child has margins
- An element without content, padding, border, and height

Let's explore these cases:

Adjacent siblings:

In this case, the first element might have a margin of 10px (on all sides, let's say), and the second one has 5px (or 20px - the values don't matter).

CSS will collapse the margins and only add the bigger one between the elements. So if we got margins of 10px and 5px, a 10px margin would be added between the elements

A parent with children that have a margin:

To be precise, the first and/ or last or the only child has to have margins (top and/ or bottom). In that case, the parent elements margin will collapse with the child element(s)' margins. Again, the bigger the margin wins and will be applied to the parent element.

If the parent element has padding, inline content (other than the child elements), or a border, this behavior should not occur. The child margin will instead be added to the content of the wrapping parent element.

An empty element with margins:

This case probably doesn't occur that often but if you got an element with no content, no padding, no border, and no height, then the top and bottom margin will be merged into one single margin. Again, the bigger one wins.

CSS display property

Depending on the element's `display` property, its box may be one of two types: a block box or an inline box.

The box model is applied differently to these two types.

First, let's understand the `display` property.

The display describes the way the elements behave in the document flow, that is, the elements' positioning on the website:

- Each block-level element always uses the entire space available in the line where it is positioned, and each block-level element also starts in a new line.
- Each inline element uses only the space needed to display the actual content, and after this content is displayed, the following inline element is placed next to it without going to a new line.

Inline box

Inline boxes are laid out horizontally in a box called a line box. If there isn't enough horizontal space to fit all elements into a single line, another line box is created under the first one. When an inline box is split across

more than one line, it's still logically a single box. This means that any horizontal padding, border, or margin is only applied to the start of the first line occupied by the box, and the end of the last line. Inline boxes completely ignore the top and bottom margins of an element.

The horizontal margins display just like we'd expect, but this will not alter the vertical space.

If we change the margin to padding with blue color, we'll notice that it'll display the blue background; but it won't affect the vertical layout of the surrounding boxes and not add any space.

If you want to play with the vertical space of a page, you must be working with block-level elements or change the display to block.

Block box

Block boxes always appear below the previous block element. This is the "natural" or "static" flow of an HTML document when it gets rendered by a web browser. Here the margin, border, and padding can be applied to alter the width and height of the content as required. We can override the default box type of HTML elements using the CSS display property:

```
em, strong {
 background-color: #B2D6FF;
 display: block;
}
```

Inline-block

The `inline-block` display property changes the behavior of block elements to prevent them from adding a line-break after the element, so the elements can sit next to other elements.

You can set the `width` and `height` of the inline-block element like a block element, and also the top and bottom margins/paddings are respected.

Inline	Block	Inline-block
Don't start on a new line	Start on a new line	Don't start on a new line
Block elements to the left and right of an inline element will not add space to the top or bottom padding or margin of an inline element.	Block elements to the left and right of a block element will add space to the top or bottom padding or margin of an inline element.	Block elements to the left and right of an inline-block element will add space to the top or bottom padding or margin of an inline element.
Can be changed by using display property as block or inline-block for any element.	Can be changed by using display property as inline or inline-block for any element.	Can be changed by using display property as inline or block for any element.

CSS overflow

The overflow CSS property specifies what to do when the content is too large to fit in its containing element block. It is a shorthand for the `overflow-x` and `overflow-y` properties, which can be specified

directly and separately.

It can take the following values:

- `Visible`: The content overflows and shows beyond the container.
- `Hidden`: The content is truncated, and extra content is not visible.
- `Scroll`: Adds scroll bars to view the clipped content.
- `Auto`: Adds scroll bars only if the content overflows beyond the container.

Positioning elements

The position property deals with how the elements are positioned on the page.

The static value(default) implies that the elements follow the regular document flow, and there are no deviations from the normal flow.

Following the document flow means that the elements are added in the order they appear in the document (HTML).

Sometimes it becomes necessary to break the normal flow. For example:

- A sidebar to the right
- A fixed navigation bar on the top
- A background image for all the contents of the page

If you want to change the normal flow and position elements differently, you can make use of the different options of the position property available:

- fixed
- absolute
- relative
- sticky

Another option to break the normal flow is using the `float` property. The `float` property positions the element to the left or right, and the other elements flow around it.

Whenever an element breaks the normal flow, other elements totally ignore that element as if it is not in the flow.

We will see each one of the options to achieve this in detail.

Fixed

This fixes the position of the element with respect to the complete viewport:

- This can be applied to both block-level and inline elements.
- This can be used to fix a navigation bar at a specific location.
- It always stays in the same place, even if the page is scrolled.
- Using `top`, `bottom`, `left`, and `right` properties, we can give the position where we want to fix the element.

For example, for navigation bar on `top-left`, `top:0; left:0; position:fixed`;

If margin is 0, the elements may not require the extra properties (top,left) for a fixed navigation bar on top.

Absolute

This fixes the position of the element with respect to the nearest positioned (any element with a position property other than static) ancestor.

If no positioned ancestor exists, it positions with respect to the HTML document body and moves along.

Relative

This property does not have any impact on its own, and it keeps the element in the normal flow.

If the positioning attributes, top, bottom, left, and right are applied, then its position changes relative to its position in the normal flow.

It can make a normal flow element positioned so that you can apply z-index, and also its children can be positioned using the absolute property.

Sticky

This is a combination of relative and fixed.

Just by adding the property, there is no change in the behavior of the element like `relative` property. If we specify a top property, the element behaves as fixed as soon as the element reaches the specified distance from the viewport. The element stops being fixed as you scroll to the end of the content inside the containing parent element. Sticky-positioned elements are only sticky within their parent element.

This is not very commonly used as the browser support is partial.

Stacking context

We positioned the elements using the x and y coordinates so far. The stacking concept looks into the 3D aspect of positioning.

The stacking context is a three-dimensional conceptualization of HTML

elements along the imaginary `z-axis` relative to the user, who is assumed to be facing the viewport or the webpage.

HTML elements occupy this space in the order of priority indicated by the element attributes.

By default, when the z-index property is not specified on any element, elements are stacked in the following order (from bottom to top):

- The background and borders of the root element
- Descendant non-positioned blocks, in order of appearance in the HTML (position: static in order of appearance)
- Descendant positioned elements, in order of appearance in the HTML (position: absolute, relative in order of appearance)

If you want to create a custom stacking order, you can use the z-index property on a positioned element.

The `z-index` property can be specified with an integer value (`positive`, `zero`, or `negative`), which represents the position of the element along the z-axis. The default value auto equates to value 0 levels on the z-axis, which is the default level for all elements. The z-index applied to a static (default) positioned item does not have any impact.

For example, if we have to make an image as a background image, then I would need to apply two main properties to the class to move it behind the other elements:

```
.background{
url :('image.jpg');
position: fixed;
z-index:-1;
}
```

Using the different variations of position property, you can place your content on the screen as needed, but it is not very flexible, and you need to really use different combinations of properties for spacing and alignments.

With CSS Flexbox, placing items in a specific direction, either x-axis or y-axis, and aligning with respect to these axes has become very easy. Next, we will explore this new CSS layout, which is a true savior of styling and positioning.

CSS Flexbox

The CSS Flexbox layout applied to the container makes it flexible to fill up or shrink in the available space.

There are two parts of CSS Flex layout which has its own set of properties:

- **The Flex Container:** This is the containing block that you declare to

be flexible by using the properties applicable at the container level.

- **The Flex Items:** These are the items within the containing block that gets displayed within the container with different orientation and alignment, which can be controlled by different properties at both container and item level.

The following diagram shows the flex container and the flex items contained within it:

Figure 3.6: *The CSS Flexbox Layout*

Now that you understand about the container and items let's look at the properties which can be applied at both levels and what impact they have.

Flex container properties

- `display: flex`

  The first property converts the container into Flexbox layout so that the items within it can be laid out using the other flex properties. All child elements within the container become flex items on applying this property.

- `flex-direction`

  The flex layout provides the capability to place and aligns the flex items along its two axes, which are called the main axis and the cross axis(which is perpendicular to the main axis). The flex-items are placed along the main axis, and they can be aligned with respect to both the axes using the different flex-properties. Using the flex-direction property, you can set the direction of these twoaxes.

  The `flex-direction` property can be as follows:

  - **row:** The main axis is the x-axis(along the row) in the direction of the inline element, and the cross axis is the *y-axis* (along the column) The start will be based on the default direction of the language (For English it will be left to right, for Arabic it will be right to left).
  - **row-reverse:** The opposite of the row direction(start and end gets interchanged)
  - **column:** The main axis is the *y-axis* (along the column), in the direction of the block from top to bottom, and the cross axis is along the row.
  - **column-reverse:** The main axis is along the column, in the opposite direction of the block from bottom to top. The cross axis will remain along the row.

This property can be changed based on the direction in which we want to

place the flex-items. We can place the items as shown below using this property:

Figure 3.7: *Flex-direction property main-axis directions for ltr language like English*

- `flex-wrap`:This property controls whether all flex-items should fit in a single line or wrap to the next line.

  The values for this property include:

  - `nowrap`: All fit in a single line without any wrapping
  - `wrap`: The items will wrap to the next lines from top to bottom
  - `Wrap-reverse`: The items will wrap from bottom to top

- `flex-flow`: This is a shorthand property, which is a combination of flex-direction and flex-wrap.
- `justify-content`: This property is used to align the flex-items along the main-axis, also organizing the extra space and distributing it as needed. The values for this property include:

  - `flex-start`: Items aligned along with the start of main-axis.
  - `flex-end`: Items aligned along the end of main-axis.
  - `center`: Items aligned along the center of main-axis.
  - `space-around`: Items with space added evenly with space at the start and end of the items in the line.
  - `space-between`: Items with space added evenly with the first item at the start, and the last item at the end with no space added on either side.
  - `space-evenly`: Items are aligned with space distributed equally.

- `align-items`:

  This property is used to align the flex-items along the cross-axis, also organizing the extra space and distributing it as needed. The values for this property include:

  - `flex-start`: Items aligned along with the start of cross-axis
  - `flex-end`: Items aligned along the end of cross-axis
  - `center`: Items aligned along the center of cross-axis
  - `stretch`: Items stretched to fill up along the cross-axis
  - `baseline`: Items with text aligned along the same line

- `align-content`

  This property works with multiple lines of content in case wrap was used. The space between the content lines is managed along the

cross-axis. The values for this property include

- `flex-start`: Items aligned along with the start of main-axis
- `flex-end`: Items aligned along the end of main-axis
- `center`: Items aligned along the center of main-axis
- `space-around`: Items with space added evenly with space at the start and end of the items in the line
- `space-between`: Items with space added evenly with the first item at the start, and the last item at the end and no spacing is added on either side
- `space-evenly`: Items are aligned with space distributed equally

Flex-item properties

Additionally, there are some properties that can be applied at the flex-item level to control their behavior and alignment. The `flex-item` properties include the following:

- `flex-grow and flex-shrink:`

  These two properties take numeric values and give the proportion in which an item will grow/shrink if needed. If given as 1 for all items, all items will grow and shrink equally. If some item is given a value of 3 when compared to others being 1, it will grow/shrink three times the other items. If set to 0, the item will not shrink or grow.

- `flex-basis:`

  This will be the default size of the flex-item without taking the extra space into consideration. It will be the default width or height based on the flex-direction if set to auto.

- `flex:`

  This is the shorthand property of `flex-grow`, `flex-shrink`, and `flex-basis`.

- `align-self:`

  This property is used to apply a different alignment at the `flex-item` level along the cross-axis. If you want to apply a specific alignment different from the one applied using the align-items property, you can use the align-self property at the item level. The values for this property include

  - `stretch`
  - `center`
  - `flex-start`

- flex-end

- order:

  This property can be used the change the order in which the flex-items appear in the flex-container. By default, the value is 0, and the items appear in the order in which they are added. If you want to change the order, you can add a higher value to move it to the end, or add a negative value to move it to the beginning of the flex-items list.

  Using these properties applied to the `flex-container` and `flex-items`, you can control how elements are placed along the row or column direction with proper alignment and spacing.

  In the example below, using the flex properties, you can see how easily you can change the position, spacing, and alignment of the items.

  The HTML file:

```
<html lang="en">
<head>
<meta charset="UTF-8">
<link rel="stylesheet" href="main5.css">
<title>Flexbox</title>
</head>
<body>
<div class="flex-container">
<div class="itemA">div only</div>
<div class="itemB">w150px</div>
<div class="itemC">h200px</div>
<div class="itemD">w/h400px</div>
<div class="itemE">w150px</div>
<div class="itemF">w150px</div>
</div>
</body>
</html>
The css file
* {
  box-sizing: border-box;
  font-size: 1.5rem;
}
html {
  background: grey;
  padding: 5px;
}
body {
```

```
  background: grey;
  padding: 5px;
  margin: 0;
 }
.flex-container {
  background: white;
  padding: 10px;
  border: 5px solid black;
  height: 1200px;
  display: flex;
  flex-direction: row;
  flex-wrap: wrap; /*flex-flow: row wrap;*/
  align-items: center;
  justify-content: center;
  align-content: center;
 }
.itemA {
  background: teal;
color: white;
  padding: 10px;
  border: 5px solid black;
  margin: 10px;}
.itemB {
  background: orange;
color: white;
  padding: 10px;
  border: 5px solid black;
  margin: 10px;
  width: 150px;
  font-size: 1.8rem;
}
.itemC {
  background: pink;
color: white;
  padding: 10px;
  border: 5px solid black;
  margin: 10px;
  height: 150px;
}
.itemD {
  background: blue;
color: white;
  padding: 10px;
  border: 5px solid black;
```

```
    margin: 10px;
    width: 400px;
    height: 400px;
.itemE {
    background: greenyellow;
color: white;
    padding: 10px;
    border: 5px solid black;
    margin: 10px;
    width: 150px;
}
.itemF {
    background: cyan;
color: white;
    padding: 10px;
    border: 5px solid black;
    margin: 10px;
    width: 150px;
}
```

The output looks as below:

Figure 3.8: *Flex example*

With little variation of flex-direction to column, the output looks like the following image:

Figure 3.9: *Flex example*

You can check out the different variations of the different flex properties on this example, which is also included in the code bundle.

Responsive design

Before concluding on the overview of CSS, there is one important aspect we need to look at surely. In this world of different devices, how to make web content that looks good on any screen size! That's responsiveness, and it is a key factor to consider in modern web development. This ensures the website looks good and behaves well irrespective of the screen size.

The two main tools used for responsive design include:

- Viewport
- Media queries

Viewport

This adjusts the width to the device width, which applies the pixel ratio

and translates to the CSS pixel size for the current device:

```
<meta name="viewport" content="width=device-width,
initial-scale=1.0">
```

Let's understand the different attributes we can set using the viewport tag to ensure our website adjusts to the available screen size:

- `viewport`: Tells the area for the website is the screen available
- `width`: Tells the browser of the available width so that it applies the pixel ratio and translates the hardware pixel to software pixel
- `initial-scale`: Defines the zoom level
- `user-scalable`: To allow the users to zoom in and out by setting this to yes/no,default is yes
- `maximum-scale`: Restricts the zoom level to a maximum value
- `minimum-scale`: Restricts the zoom level to a minimum value

Media queries

Media queries allow you to define conditions based on the type and width of the device. A media query is composed of an optional media type and any number of media feature expressions:

- Media types describe the general category of a device. For example, all screen, print, speech.
- Media queries are like if statements. If the condition is satisfied, one or more CSS properties can be set as per the device need. You can have more than one media query with different conditions to handle devices of different sizes.
- First, you identify the breakpoints based on the different device sizes which you need to handle. Then define the CSS for each conditionally defined breakpoint as follows:

```
@media screen and (min-width: 600px) {
.MyClass {
       width: 30%;
       float: right;
      }
}
```

Using the above media query, you can apply above styling to `MyClass` class elements for devices that have a minimum width of 600px, that is, devices like desktops and laptops.

Now the media query condition is changed to `max-width` as below:

```
@media screen and (max-width: 600px) {
```

```
.MyClass {
        width: 100%;
        }
}
```

Using this media query, the condition now applies to devices with a maximum width of 600px, like mobile phones, where the elements with class name `MyClass` will take up the entire width of the screen.

When starting your development, you can choose to start with any of the following approaches depending on the project requirements:

- **Mobile First:** In this approach, you style your application considering a mobile screen and later define media queries to handle the styling requirements for bigger screens(using `min-width` conditions).
- **Desktop First:** In this approach, you start with styling your application considering a bigger screen and later define media queries to handle the styling for smaller mobile screens (using `max-width conditions`).

Using different breakpoints, you can apply different styling rules to handle different device sizes.

Conclusion

The beauty of your websites will be determined by your hold on CSS. In this chapter, you were introduced to the different aspects of writing CSS rules, selectors, combinators, rules of specificity, different properties to position, and modify layouts. You can improve your CSS skills by lots of practice and application, which gives you the comfort of making the right choice, what to use, and when. With your website looking beautiful, you need to make it interactive and dynamic. In the next chapter, we will look into this aspect of web development, making your web smart using JavaScript, the brain of the web.

Questions

1. The ___ property specifies the stack order of an element (which element should be placed in front of, or behind, the others).

   A. z-index

   B. display

   C. img

   D. None of the above

   Answer: A.

Z-index is used to stack the elements on the 3D.

2. Which of these combinators are correctly matched?

   A. div + p - adjacent sibling selector
   B. div > p - child
   C. div ~ p - general sibling selector
   D. All of the above

   Answer: D.

   All the above combinators are matched correctly with their corresponding syntax.

3. Which of the following CSS properties DOES NOT influence the box model?

   A. Content
   B. Margin
   C. Outline
   D. Border

   Answer: C.

   The outline is not a part of the box model.

4. What is the positioning context of an element after position: fixed was applied?

   A. The containing block
   B. The HTML element
   C. The viewport
   D. The element itself

   Answer: C

   The position fixed will position the element with respect to the complete viewport.

5. The CSS property specifies if an element's background, whether a <color> or an `<image>`, extends underneath its border.

   A. background-size
   B. background-position
   C. background-clip
   D. background-origin

   Answer: C.

```
Chapter-4-JS ▸ index.html ▸ html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>sHost</title>
    <link rel="shortcut icon" href="favicon.png">
    <link rel="stylesheet" href="main.css">
</head>
<body>
<div id="name"></div>
<div id="details"></div>
<button onclick="myJSFunctionIn()">Inline </button>
<button onclick="myJSFunction()">External File JS</button>
<script>
    function myJSFunctionIn() {
     document.getElementById("name").innerHTML = "This JavaScript is within the same file within script tags ";
    }
</script>
<script src="myJScript.js"></script>
</body>
</html>
```

```
Chapter-4-JS ▸ myJScript.js ▸ myJSFunction
function myJSFunction() {
    document.getElementById("details").innerHTML = "This is from external JavaScript file ";
}
```

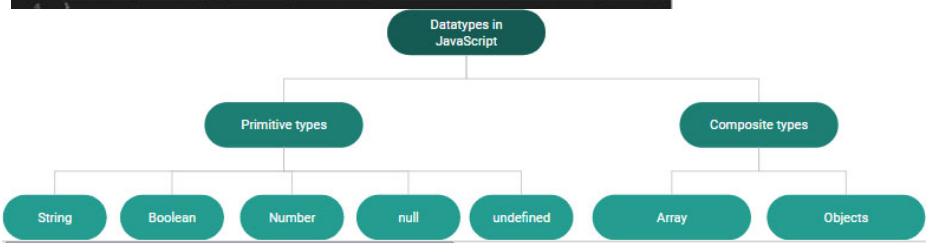
Inline | External File JS

This JavaScript is within the same file within script tags

Inline | External File JS

This JavaScript is within the same file within script tags
This is from external JavaScript file

Inline | External File JS




```
var global1=11;
let global2 =12;
const global3 = 13;

function outer(param1){
var local1=10;
let local2=20;
console.log("Outer Global");
console.log(global1 + global2+ global3);

function inner(param2){
var global1 = 30;// can be re-defined using same name
//let global2= 40;// error

if(param2){ //new block
let block1=50;
const block2=60;
console.log("Inner Global");
 console.log(global1 + global2+ global3);
console.log("Local");
console.log(local1 + local2 + param1 + param2);
console.log("Block");
console.log(block1 + block2);
}

//console.log('Outside Block',block1 + block2);// error
}

console.log(inner(21)); }

outer(31);
```

Uncaught SyntaxError: Identifier 'global2' has already been declared

Uncaught SyntaxError: Invalid or unexpected token

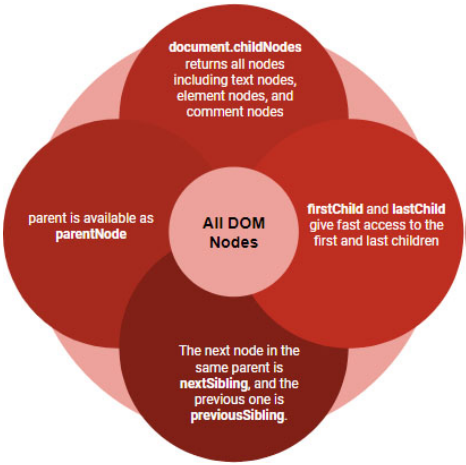
The global scope variables accessible anywhere in the code

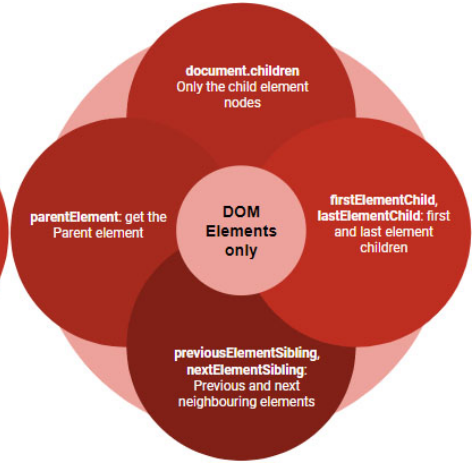
The function scope local variables including the parameters accessible within the function including the inner function

The inner function variables not accessible in the outer function, but can access the outer function variables and parameters. The global var global1 is being re-declared with new value which overrides old value. let/const global variables cannot be redeclared.

The block scope variable only accessible within the block, cannot be accessed outside block

```
Outer Global
36
Inner Global
55
Local
82
Block
110
```


document.childNodes
returns all nodes including text nodes, element nodes, and comment nodes
parent is available as parentNode
All DOM Nodes
firstChild and lastChild give fast access to the first and last children
The next node in the same parent is nextSibling, and the previous one is previousSibling.


document.children
Only the child element nodes
parentElement: get the Parent element
DOM Elements only
firstElementChild, lastElementChild: first and last element children
previousElementSibling, nextElementSibling: Previous and next neighbouring elements

CHAPTER 4

JavaScript Programming: Making the Application Interactive

"Great web design without functionality is like a sports car with no engine."

- Paul Cookson

"Design is not just what it looks like and feels like. Design is how it works."

- Steve Jobs

Now that we know how to add and beautify web content using HTML and CSS, we next have to make this content live and interactive using JavaScript programming. JavaScript is the brain behind all the beauty of the web content. JavaScript defines how user interaction will be handled, what actions will happen, and what will be the next state of the application. In this chapter, you will be introduced to the basics of JavaScript, and you will get into the details of different aspects of JavaScript programming in the following few chapters.

Structure

- Introduction to JavaScript

- Building blocks of JS
- More about functions
- Arrays
- Working with DOM

Objective

After studying this chapter, you will learn the basics of JavaScript and how to incorporate interactiveness and dynamic behavior to make your HTML pages smarter.

Introduction to JavaScript (JS)

JavaScript (JS) is a light-weight, cross-platform, object-oriented computer programming language, which was developed to add logic and behavior to the web. In Chapter 1, we saw a brief history of how JavaScript was born and how it continues to grow with every ECMAScript edition.

Using JS, you can make your website dynamic and interactive. The power of JavaScript also lies in the fact that it can be used for both client-side and server-side programming as it can execute in the browser or on the server, or on any device that has a special program called the JavaScript engine.

In *Chapter 3,* we saw how the browser parses HTML and CSS content separately, and then the document is constructed and displayed:

- So how does JS fit in here?
- What if we have some JavaScript associated?
- And firstly, how do we associate JavaScript into the HTML/CSS code?

Let's try to answer these questions one by one.

JavaScript is a scripting language and it can be included in the HTML using the <script> tag which can hold the direct JS code embedded within HTML either in the head or the body section.

```
<script>
function myJSFunction() {
 document.getElementById("name").innerHTML =
 "JavaScript";
}
</script>
```

It can also be written as a separate external file and included as a source in head or body as shown as follows:

```
<script src="myJScript.js"></script>
```

Where `myJScript.js` is an externally defined file with JS code. The second approach of having an externally defined JS file is the recommended way as it is cleaner and keeps the code segregated.

By including the above piece of JavaScript, you can change the content of the HTML, as shown in the below example. The left side shows the HTML file with the embedded JavaScript code and also referring to an external JavaScript file. The right side shows the html page rendered initially, and then the content gets added as the user clicks the two buttons one by one.

Figure 4.1: *Example of HTML with JavaScript*

This is the most basic example, but this shows the power of JS to make changes to the HTML content dynamically, as shown on the click of the button here. On the click of the button, we were able to include content in the web page output. Now that you understand how you can include JS code in your web content, you will learn about various aspects of JavaScript and how you can put that to use. Let's first look at the steps showing how the JS engine works?

- The engine parses the script
- Then it compiles the script to the machine language
- Finally, the machine code is what runs, which is pretty fast

Also, the engine applies optimizations at each step of the process to ensure that there are no lags, and the code is fully optimized.

JS adds life to the webpage, adds interactivity, etc. for example, to name a few things:

- Handle user actions like button clicks, scrolls, mouse movement, and so on
- Change attributes, styles related to elements in response to some selective behavior
- Add cookies, save session-related and user-related data locally

Let's delve into learning JavaScript, which is the backbone of the web. When learning a new language, always start with the basics.

Building blocks of JS

You will now learn about the basic syntax and building blocks of the JavaScript scripting language.

The building blocks of JS mainly include the following:

- Variables
- Data Types

- Statements
- Functions

Variables

Variables are identifiers used to store the data and information related to the application. Variables are needed in any programming language to hold values of different types, computation results, etc.

Variables can be defined using the below keywords (this includes the latest approach of defining variables as per ES6):

- `let`: This is the recommended way of defining variables that remain valid in the block in which it is defined, and its value can be changed and reassigned. A block is defined as a set of lines of code enclosed within a pair of parentheses, so it could be a `for` loop, an if statement, a function, any block of code within a pair of parentheses.

  ```
  let name;
  let age;
  let num1,num2,num3;
  ```

- `const`: This is used whenever the containing value is constant and will not undergo change. This is also block scoped. This is the recommended way of defining if you are sure that your variable value should not be changed as it will ensure that it does not get overwritten even by mistake.
  ```
  const pi=3.14;
  ```
- `var`: This is the traditional way of defining variables in JS, which is not block scoped but function scoped.
  ```
  var dummy;
  ```

For a basic understanding of how scope and var work traditionally, we will briefly explain the concept of scopes in JavaScript.

Scopes

Scope in JavaScript tells us which variables will be accessible at a given point.

There are two kinds of scope – global scope and local scope.

Global scope

Any variable which is declared outside any function is accessible anywhere in the code, even in the functions and is in the global scope.

```
const global = 'Hi! I am Global';
```

You can define global variables, but it is recommended to avoid their use as it may result in name collisions if multiple variables are given the same name. Especially with the use ofvar for declarations, you will not get any error if we declare and use the same variable name multiple times. So better to avoid global variables.

Local scope

Local variables are accessible only in a specific part of your code. The local scope can be of two types as follows:

- **Function scope:** When a variable is declared within a function, it is accessible only within the function. You can't access this variable once you are out of the function.

```
function HelloWorld () {
 const hello = 'Welcome to JS!';//this is function
 scoped
 console.log(hello);
}

HelloWorld (); //'Welcome to JS!'
console.log(hello);// This will give error hello is
not defined
```

- **Block scope:** When a variable is declared using the `const` or `let` keyword, within a block of curly brace ({}), it is accessible only within that curly brace.

```
{
 const hello = 'Welcome to JS!';//this is block
 scoped
 console.log(hello);
}
console.log(hello);// This will give error hello is
not defined
```

Multiple values can be declared in a single statement. An initial value can be provided at the time of declaration or later, which is called variable initialization. If initialization is not done, it takes the value undefined.

Some things to remember when defining variable names:

- The names should make proper sense related to their purpose.
- It can contain $ (dollar) and _ (underscore) symbols but not - (hyphen)
- Short but descriptive. Not, single letter names like a, b, c.
- Use camelCase, which starts with the lower case but uses Capitals for

any new words in the name like `firstName, lastName`.

Let's summarize the differences of let versus const versus var in the next table:

	Const	Let	Var
Block scoped	Block scoped.	Block scoped.	Not block scoped.
The variable can be used only within the block of code bound by parentheses within which it is declared	The variable can be used only within the block of code bound by parentheses within which it is declared.	The variable can be used only within the block of code bound by parentheses within which it is declared.	The variable can be used anywhere within the function bound by parentheses within which it is declared.
Gives a Reference Error if trying to access the variable before it is declared	Gives a Reference Error if trying to access the variable before it is declared.	Gives a Reference Error if trying to access the variable before it is declared.	Gives a Reference Error if trying to access the variable before it is declared.
Cannot be reassigned to a different value.	Cannot be reassigned to a different value.	Can be reassigned to a different value.	Can be reassigned to a different value.

Table 4.1: *let versus const versus var*

Data Types

After having defined variables in JavaScript, next, let's look at the different types of values which can be held in variables and used in JS. JavaScript has some main data types, but no specific data type is tied to a variable. JS is a loosely and dynamically typed language, which means that there are data types in the language, but variables are not bound to any of the types at the declaration time. This means that when we are declaring the variables using let/const/var, we do not specify any data type associated with the variable. The data types indicate the different types of values which can be used and manipulated in JavaScript.

The main data types include the following:

Figure 4.2: *Data Types in JavaScript*

- **Number:** A primitive data type to handle numbers of any type integer or floating-point. You can perform mathematical operations on numbers (+,-,*,/,%):

  ```
  let pi=3.14;
  let infiniteNum= 12/0; //This will return infinity
  const notANum= 'here'/2; //This will return NaN
  which stands for Not a number, an invalid
  expression
  ```

- **String:** Another primitive data type which may include one or more characters enclosed within quotes. Both single and double quotes can be used to bound the string:

  ```
  let name="JavaScript";
  let fullName='JavaScript'+'Rocks';
  ```

  Another type of quotes which can be used are backticks (‘’). They help in extending functionality and allows to embed variables and expressions into a string by wrapping them in ${...} which gets evaluated to form the complete string as a result:

```
let greeting = `Hello, ${fullName}!` ;// the
variable fullName value will get substituted and
assigned
console.log(greeting);
```

- **Boolean:** A primitive data type which takes only two values `true/false`, which represent the yes/no case.

```
let showFlag=true;
let BigNum = num1>1000; // condition can return
true or false as the result
```

- null is used for any unknown value. It has a single special value null, which indicates `nothing", empty or value unknown.`
`let yearOfManufacture = null;`
- **undefined:** Another single special value which indicates that the value is not assigned. It can also be used to assign the value undefined as follows explicitly:

```
let num1;
console.log(num1); //undefined
let num2=10;
num2=undefined;
console.log(num2);//undefined
```

- **Object:** This is complex data for handling complex data structures, including data of different types.

  A single object can include data of different types as follows:

```
var person={
 name:"Peter",
 age :24,
 designation:"Software Engineer",
Allocated: true}
```

- **Arrays:** This is a subtype of an object to handle the list of items of the same primitive type:

```
var cities= ["Hyderabad", "NewYork", "Calgary"];//
here all the 3 values are strings
```

- **Symbols:** This is new to JavaScript included as part of ECMAScript 2015. It is a unique and immutable primitive value that can be used as an identifier or the key of an Object property.

Working with Data Types

When working with Data Types, we need to be aware of certain useful operators and methods.

One such method is used to determine what is the type of data held by a variable at runtime. This is useful as we do not tie a data type to a variable when it is declared, but we may need to perform some operations on the variable based on the type of data it receives dynamically. The typeof operator can help find the type of your variable.

typeof operator:

The typeof operator can be applied to any value, and it returns a string indicating the data type of the value.

It can be used as typeof operand or `typeof(operand)` as shown as follows:

```
console.log(typeof 19); //number
console.log(typeof "JavaScript"); //string
console.log(typeof true); // boolean
console.log(typeof undefined); // undefined
```

JavaScript Data Type conversion:

Sometimes it becomes necessary to convert the variables from one type to another. JavaScript variables can be converted to another data type using the following methods:

- JavaScript function
- Automatically by JavaScript itself - JavaScript performs this conversion whenever it finds a mismatch, but it may not be what we exactly want, so it is better to handle explicitly using functions

Some of the common type conversions using functions:

- To convert to a string: Any variable can be converted to a string using the below options:
  - `String()`: The global method String() can convert numbers, booleans, literals, expressions, dates to strings.

    ```
    String(false) // returns "false"
    String(Date())// returns current date in string format
    ```

- `ToString()`: This method does the conversion on any number and converts to string:

  ```
  (498).toString() // "498"
  ```

Operators

Operators operate on one or more operands to give a result. Operators will

be applied to the variables to perform value manipulations and computations. Based on the number of operands they work upon, operators can be unary, binary, or ternary.

For example:

```
2++; // Unary Operator
2 + 3; // Binary Operator
(4 > 3) ? true : false; //Ternary Operator
```

Types of operators:

Following listed are the various operators applied to variable to form expressions:

- **Arithmetic Operators:** Perform arithmetic computations. Like `+`, `-`, `*`, `/`, `%`, `++`, `--` as shown as follows:

```
let num1 = 6;
let num2 = 4;

num1 * 4; // Multiplication
num1 / num2; // Division

let rem= num1 % num2; // Modulus
num1++; // Post-increment
++num1; // Pre-increment
num1-- // Post-decrement
--num1 // Pre-decrement
```

- **Assignmentoperators:** To assign values from the right expression to be evaluated and assigned to the variable on the left side of the equal sign(=) as shown as follows:

```
var num = 4; // Assignment Operator
num += 2; // Addition Assignment, same as num = num
+ 2;
num -=1; //Subtraction Assignment
num *= 5; //Multiplication Assignment
num /= 2; // Division Assignment
num %= 2; //Modulus Assignment
```

- **Comparison operators:** To compare different values and expressions on how they stand with respect to each other as shown as follows:

```
5 === "5" //Strict Equality, false
5 == "5" //Equality, true

5 !== "5" //Strict-non-equality, true
5 != "5" //Non-equality, false

5 < 5; //Less than, false
```

```
5 <= 5 // Less than or equal to, true

5 > 4 // Greater than, true
5 >= 6 // Greater than or equal to, false

(5 == 5) ? true : false; //Ternary, true
```

- **Logical operators:** To add logical conditions (And, Or, Not) to formulate expressions as follows:

```
var x = 4;
var y = 2;

(x < 10 && y > 0); // and, true - this is true only
if both conditions are true

(x == 5 || y == 5); //or, false - true if atleast
one of the conditions is true

!(x == y); // not, true
```

Comments

Comments form an integral part of any programming language and are included to make the code clearer and easy to understand. It is considered a good programming practice to add enough comments so anyone can understand your code just by looking at it with the supporting comments.

Comments can be added as:

1. **Single line comments:** These are preceded by // and contain the comment in one line. Any executable line of code can also be commented so that it is skipped using // and converting it into commented code.

```
//initializing the data
let num1=100;
```

2. **Multi-line comments:** These are enclosed within /* and*/. Any code within these will not be executed, and this is used to add comments spanning across multiple lines:

```
/*The function is used to generate the fibonacci
series
Of numbers */
 function fibbo(){
}
```

Statements

A JavaScript statement is made up of variables, values, operators, expressions, keywords, and comments. It could be the following:

- Variable declaration
  `let num1, num2;`
- Assignment
  `let sum = num1 + num2;`
- Return statements
  `return sum;`
- Break/Continue
- Debugger

A JS statement should be terminated with a semicolon. Though it is optional, it is a good practice to use it to avoid any surprises later.

A block of statements could be related and executed together in the following forms:

- **Conditional statements:** Blocks of code getting executed conditionally if a given condition is `true`. This includes the following statements:
  - **if..else statements:** Conditionally execute code based on the expression being true as shown as follows:
    ```
    If (num1>num2)
    return num1; // this will be executed if num1 is
    greater than num2
    else
    return num2; // this will be executed if num2 is
    greater than num1
    ```
  - **switch case statements:** Conditionally execute code based on the value match as shown:
    ```
    switch(x) {
     case 'value1': // if (x === 'value1') this case
     will be picked
      [break]
     case 'value2': // if (x === 'value2') this case
     will be picked
      …
      [break]
     default:// this will be picked if no other
     condition is true
      …
      [break]
    }
    ```
- **Iterative statements:** Block of code getting executed for multiple numbers of times based on the loop condition. Loops are used to

perform a repetitive task in the program code. They can be entry-controlled or exit-controlled depending on the placement of the loop condition:

- for loop: An entry controlled loop with the following syntax:

```
for (initialization; test condition; updation) {
 // … loop body …
}
Example : for(i=0;i<10;i++)
{
console.log(i);
}
```

- **While loop:** Another entry controlled loop with the following syntax:

```
Initialization;
while( test condition) {
 //body of loop
 Updation ;
}
```

Example:

```
i=0;
while(i<=10){
 console.log(i);
 I++;
 }
```

- **do…while loop:**

```
Initialization;
do {
 //body of loop
 Updation;
} while(test condition);
```

Example

```
i=0;
do{
 console.log(i);
 I++;
 }while(i<10);
```

- **try…catch statements:** To handle any errors in the `try` block of code, the `catch` block has the logic of error handling, it is as shown:

```
try{
Logic…
}
catch(error){
```

```
Error handling logic…
}
Example:
try {
  throw {
  error : " Throwing error"
  };
 } catch(error) {
     console.log(error);
}
```

- **Function:** A block of code performing some action. We will learn more about functions in the next section.

Functions

We have always seen people repeating the same code again and again, which raises the need for putting the common code in a single place and reuse it. Functions help us achieve that in different ways.

What is a function?

A function is a code block that is designed to work together to perform a specific task. The task to be done by the function is defined once and can be invoked or executed in the form of a function, a number of times.

A simple explicit function can be defined is shown as follows:

```
function calculateDistanceTravelled( startPos,
currentPos ) {
 var distance = Math.sqrt( Math.pow(currentPos.xPos –
 startPos.xPos, 2) + Math.pow(currentPos.yPos –
 startPos.yPos, 2));
 return Math.ceil(distance);
}
```

Here you have done the following:

- `function` keyword is used to explicitly define and give a name to the function as the name, `calculateDistanceTravelled`
- The function accepts two parameters (JavaScript objects)
- A local variable is used inside the function to hold the calculated distance value
- Logic to apply ceiling to the calculated distance
- Finally, return a value from the function to indicate the end of execution flow and transfer of control back to the caller

- Note that if the `return` statement is not explicitly provided, then undefined is returned as a value.

More about functions

In the previous section, we defined a function. Now we will use it by invoking it. For this we will have to pass the required parameters to the function call as shown in the following code block:

```
var startPos = { xPos : 20, yPos : 20 };
var currentPos = { xPos : 40, yPos : 40 };
document.write("You have travelled : " +
calculateDistanceTravelled(startPos, currentPos) + " KM
");
```

Function as expression

We used the Pow method of the Math object to define the function in the example earlier. Let's define an explicit function like square to calculate the square of a number, using a function expression as shown.

```
var square = function(x) {
return x * x;
};
function calculateDistanceTravelled(startPos,
currentPos) {
 var distance = Math.sqrt( square(currentPos.xPos -
 startPos.xPos) + square(currentPos.yPos -
 startPos.yPos));
 return Math.ceil(distance);
}
var startPos = { xPos : 20, yPos : 20 };
var currentPos = { xPos : 40, yPos : 40 };
document.write("You have travelled : " +
calculateDistanceTravelled(startPos, currentPos) + " KM
");
```

What did you do?

You just created a new function as an expression and assigned it to a variable named square. Later we used this variable as a function inside the `calculateDistanceTravelled` function definition wherever we wanted to use the logic to calculate the square.

We will now try out some scenarios of the code:

- Place the function invoking code (that is, the `document.write` statement) above the function declaration. What do you think should

happen?

- Place the actual function call (that is, the `document.write` statement) before the square function expression. What do you think should happen?

You may be surprised to see that while the scenario#1 has no impact on the outcome, after the scenario#2, you start seeing an error. Following block shows the modified code for scenario#2:

```
var startPos = { xPos : 20, yPos : 20 };
var currentPos = { xPos : 40, yPos : 40 };
document.write("You have travelled : " +
calculateDistanceTravelled(startPos, currentPos) + "
KM");
function calculateDistanceTravelled(startPos,
currentPos) {
 var distance = Math.sqrt( square(currentPos.xPos -
 startPos.xPos) + square(currentPos.yPos -
 startPos.yPos)); return Math.ceil(distance);}
var square = function(x) {
return x * x;
};
```

You start getting the following error, where `functionBasics.js` is the file which contains all the JavaScript code:

```
Uncaught TypeError: square is not a function
 at calculateDistanceTravelled (functionBasics.js:8)
 at functionBasics.js:4
```

Let's understand this error:

- The concern is not about the definition of `calculateDistanceTravelled`
- The concern is that square function has not been defined

Function declaration versus function expression

In the last section, you saw that you could use both the function declaration or function expression to define a function, which can be invoked and used.

However, there are some differences between these two approaches, as is evident from the error you got in scenario#2. They are useful and applicable in different use cases, and hence they do need to exist.

The main difference is the way the browser parses the function code:

- Before executing any code, the browser first looks for function

declarations and keeps them in one place

- Then it executes the rest of the code sequentially
- The function expression gets executed during the second pass

Now let's look at the error that you observed in the previous example.

The following statement invokes `calculateDistanceTravelled` function, which in turn uses square function:

```
document.write("You have travelled : " +
calculateDistanceTravelled(startPos, currentPos) + "
KM");
```

The square function gets defined as an expression at the later part of the execution and is not known to the browsers until then. Hence, it says *square is not a function.* A function declaration is already recognized before regular execution starts.

Following table summarize the differences between the function declaration and function expression:

Function Declaration	Function Expression
Parsing by browsers for declarations is first run	The code gets executed during the second pass (that is, at the execution time).
They don't have any variable assigned to them	They have a variable assigned to them
The function name is used to invoke the function	The variable name is used to invoke the function
Function declarations are not assigned to any variable	Function expressions are assigned to a variable
Function declarations are used in statements	Function expressions are used in expressions
The function gets created before the rest of the code gets executed	The function gets created only when the interpreter reaches the line of code

Table 4.2: *Function declaration versus function expression*

Now that you understand the basics of function, let's learn about different aspects of functions.

Different aspects of functions

Let's now go through and understand the different aspects of functions.

Parameter and Arguments

The values known as parameters are the ones that are defined as a part of the function declaration. The value s which are used when you invoke or call the function are called **arguments**.

We can pass zero or more parameters to the function. Also, while invoking the function, we can provide zero or more arguments. You may like to take

note of the following:

- JavaScript will go ahead with the execution, even if you will not pass the exact number of arguments
- How one handles the parameters in the JavaScript function is totally on the developer
- Each parameter acts like a local variable inside the body of the function
- If you pass more arguments than expected, then the additional arguments will be ignored
- If you pass fewer arguments than what is expected, then for the remaining arguments an undefined value will be considered
- There is no type checking in arguments, and hence anything can be passed

Parameters get passed as values. JavaScript passes arguments using the pass-by-value approach:

- Whenever you pass arguments, firstly, the values get copied, and thereafter they get assigned to the corresponding parameters. Any change made to the parameter value will impact only the parameter within the function but not have any impact on the external variable (from the outer scope), which you used to pass to the function.
- Scopes in functions: Functions cannot access variables in each other's scopes even if one function may be invoked in another.
- Nested scopes: When a function is defined inside another function, the inner function can access the outer function's variables. This behavior is called lexical scoping, and the inner function is called a closure. The outer function cannot access the inner function:

```
function outer () {
 const outerVar = 'The outer variable!';
 function inner() {
  console.log(outerVar);
 }
 return inner();
}
outer();
```

Play with functions and variables to see in action, what is in scope, and what is not, as the following example:

Figure 4.3: *Global and local variable scopes in functions and blocks*

Arrays

Arrays are a list of values that are represented with an index starting from 0. These are commonly used data type to handle the list of values. Most often, you will have a list of data to handle, and that's where arrays are very useful as you can handle the entire list in a single variable. They also provide order to your data list with the use of an index. The list can be any data type like numbers, strings, Booleans, and also objects.

In this section, you will learn about different ways to work with arrays.

The value at any position can be accessed using the index as follows:

```
let cities= ["Hyderabad", "NewYork", "Calgary"];

console.log(cities[0]); // Hyderabad
console.log(cities[1]); // NewYork
```

To define a variable of type array, we can use the following:

1. Initialize with an empty list represented by [] (empty square brackets) or a list of values within square brackets as shown:

   ```
   let names=[];
   let names=['JS','HTML','CSS'];
   ```

2. An array can be defined using the new keyword with an array type and passing the initial list of values. This is not a recommended approach as it may lead to some unexpected behavior:

   ```
   let names= new Array();
   let names= new Array('JS','HTML','CSS');
   ```

Using the first approach is the recommended way of defining arrays, so we will only be using that. Once defined, you can access the specific values using the array name followed by the index enclosed in square brackets like the following:

```
console.log(names[0]);
names[1]= 'HTML5';
Working with arrays
```

Let's now learn about the different ways we can use arrays and some of the most useful methods to work with arrays:

- **Accessing the array elements:** As mentioned earlier, the array elements can be accessed by using the index notation. For an array, the index starts from `0` as the first one, so you can compute the position of your elements from the beginning starting with `0`:
  ```
  names[4]= "React";
  console.log(names[0]);// first element
  ```
- **Calculating the length of the array:** Finding the length of the array,

which is the number of elements in the array, can be done using `array.length` as follows:

```
const len = names.length;//3
```

Length will always be one greater than the last index as you started with a 0.

To access the last element you can use the index as `array.length -1;`

In this case, names[len-1] will give the last element.

- **Determine if a variable is an array:** The typeof operator returns the object for an Array and cannot be used to determine if your variable is actually an array. We have some other ways of doing it:
  - `instanceof` operator can be used, which returns a Boolean `true/false` as shown:

    ```
    names instanceof Array // true
    ```

    This operator can be applied to other data types also in the same way.

- `Array.isArray()` is the function added as part of ECMAScript 5 which performs the check:

  ```
  Array.isArray(names);// true
  ```

- **Looping through the array elements to perform some logic:**

  a. **Traditional looping methods:**

  Using the for loop, looping till the length as follows:

  ```
  array = [ 1, 2, 3, 4, 5, 6 ];
  for (index = 0; index < array.length; index++) {
    console.log(array[index]);
  }
  ```

  Using the `do…while` loop in the same way:

  ```
  do{
    console.log(array[index]);
    index++;
  }while (index < array.length);
  ```

  b. Using `forEach` to call a function for each element of the Array as shown in the following code:

  ```
  array.forEach(function(element) {
   console.log(element);
  });
  ```

  c. **Using the map method:** The map method calls the function for each element and returns an array of the elements modified by the function.

  The following code uses the ES6 arrow method syntax, but you

can also use a normal function call:

```
let num=[10,20,30];
square= num.map((n,index)=> {
return n*n;
});
console.log(square);//[100,400,900]
```

d. **Using reduce method:** Another useful array method which also calls the function for each element of the array but returns an accumulated response where each result is passed on to the next element and finally returns a single response value.

Using a similar example with reduce to find the sum of squares will give as shown in the following code:

```
let num1=[10,20,30];
square= num1.reduce((sum,n)=> {
return sum + n*n;
});
console.log(square);// 1310
```

e. **Using filter method:** The filter method calls the function for each element, checks for a specific condition for each element and returns an array of the elements which satisfy the condition.

Similar example using arrow method syntax to return the numbers greater than or equal to 30:

```
let num2=[10,20,30, 40,50,60];
filtered= num2.filter((num)=> {
return num>=30;// true or false depending on the
element value
});
console.log(filtered);//[30, 40,50,60]
```

- **Adding new elements into the array:**

a. **Adding element using push:** Changes the array by adding elements to the end of the array and returns the length of the final array:

```
num=[1,2,3,4,5];
x= num.push(6,7,8);
console.log(num); //[1,2,3,4,5,6,7,8]
console.log(x);// 8
```

b. **Adding/joining arrays using concat:** The `concat()` method is used to join two or more arrays into a single array. It does not change the arrays being concatenated but returns a new array:

```
arr1=[1,2,3]
arr2=[4,5];
arr3=[6,7,8];
```

```
arr= arr1.concat(arr2,arr3);
console.log(arr); //[1,2,3,4,5,6,7,8]
```

- **Removing elements from the array:**

  a. **Removing last element using pop:** Changes the array by removing the last element and returns the element removed:

  ```
  num=[1,2,3,4,5];
  x= num.pop();
  console.log(num); //[1,2,3,4]
  console.log(x);// 5
  ```

  b. **Removing first element using shift:** Changes the array by removing the last element and returns the element removed:

  ```
  num=[1,2,3,4,5];
  x= num.shift(a);
  console.log(num); //[2,3,4,5]
  console.log(x);// 1
  ```

- **Modifying/replacing elements from the array:**

  a. **Adding or removing element using splice():** Changes the array by specifying the start position and number of elements to be replaced/removed and the new list used for replacing:

  ```
  num=[1,2,3,4,5];
  num.splice(2,1,11,12,13); //start at 2nd
  position and replace 1 element and add the new
  list
  console.log(num); //[1,2,11,12,13,4]
  ```

  b. **Creating a new array using elements from existing array using slice:** Creates a new aray using the start and end position specified. The original array is not impacted with this:

  ```
  num=[1,2,3,4,5];
  arr1= num.slice();
  arr2= num.slice(1,3);
  arr3=num.slice(2);
  console.log(num); //[1,2,3,4,5]

  console.log(arr1);// [1,2,3,4,5]
  console.log(arr2);// [2,3]
  console.log(arr3);// [3,4,5]
  ```

Arrays are a very useful composite data type, which you will use more and more as you get into serious programming.

Next, we will explore another power of JavaScript-DOM Manipulation.

Working with DOM

We were briefly introduced to **DOM (Document Object Model)** in the earlier chapters. To elaborate and understand further, from the MDN web docs, *"The Document Object Model (DOM) is a programming interface for HTML and XML documents. It represents the page so that programs can change the document structure, style, and content. The DOM represents the document as nodes and objects. That way, programming languages can connect to the page."*

DOM is how the web page is represented. The web page is made up of DOM nodes using which you can get access to the content and make any changes. DOM is made up of HTML elements, and CSS styles are attached to them.

Using JavaScript, if we have to interact with the page content, it will be through the DOM.

What can all be done?

- Change the HTML elements in the page
- Make changes to the HTML attributes in the page
- Make changes to the CSS styles in the page
- Add/remove HTML elements and attributes
- Handle/emit HTML events in the page

Let's see how we can do all of this and much more using JavaScript.

Each HTML element is an object as per the Document Object model. Any nested tags or content within tags are all objects and can be accessed through JavaScript as they are all objects. DOM nodes have properties and methods that allow you to modify the nodes styling properties and behavior, handling different actions on the nodes. There are 12 node types of which the most commonly used ones are the following four:

- **Document:** The first level of DOM.
- **Element nodes:** HTML-tags are the building blocks which form the DOM tree structure.
- **Text nodes:** Nodes that contain text.
- **Comments:** Content put up in the form of comments for information purposes. It won't be shown on the page, but JS can refer to it and read it from the DOM.

Traversing the DOM

If we want to access a node or element with respect to the starting tag of the structure or with respect to the current tag, the next set of methods provide access in the order of traversing the DOM. The first level node from where DOM navigation starts is the document and includes the

following set of methods to access the nodes and elements, as shown in the following diagram:

Figure 4.4: *Document methods to access child nodes and elements of the DOM*

There are two sets of methods: the first set will return all DOM nodes which satisfy the condition, the second set is with respect to element nodes only.

Nodes related methods

- **firstChild and lastChild:** This returns the first and the last child of the selected node:

```
let firstChild =
document.getElementById('refNode').firstChild;//
returns first child of an element with id 'refNode'
let lastChild =
document.getElementById('refNode').lastChild;//
returns last child of an element with id 'refNode'
```

- **parentNode:** This returns the parent node of the selected node:

```
let parent=
document.getElementById('refNode').parentNode;//
returns parent of an element with id 'refNode'
```

- **nextSibling and previousSibling:** This returns the next and previous sibling of the selected node:

```
let next =
document.getElementById('refNode').nextSibing;//
returns the next sibling of an element with id
'refNode'
let prev=
document.getElementById('refNode').previousSibing;//
returns the previous sibling of an element with id
'refNode'
```

Element related methods

The corresponding methods which return only elements but ignore any text or comment nodes while selecting the elements. The usage is the same as the node related methods:

- **firstElementChild and lastElementChild:** This returns the first and the last child elements of the selected node, ignoring any text and comment nodes.
- **parentNode:** This returns the parent element of the selected node,

not considering any text or comment nodes.

- **nextElementSibling and previousElementSibling:** This returns the next and previous sibling element of the selected node, not including the text and comment node.

Accessing elements

Using this approach, we can get direct access to a single element or multiple elements in the DOM.

The following methods can be used to access elements:

- `getElementByID`: This is used to select an element identified by a unique ID value:
```
let id = document.getElementById('myId');// this
will select the element with id=myId
```
- `querySelector`: This is used to select elements by giving a valid CSS selector which identifies an element. It returns the first element which satisfies the selector condition:
```
let qS = document.querySelector('div p');//this
selects the first p element inside the first div
```
- `getElementsByClassName`: This is used to select elements with the given class name. This can return multiple elements whichever match:
```
let classElements=
document.getElementsByClassName('red');
// this selects all elements which have the class
red
```
- `getElementsByTagName`: This is used to select elements with the given HTML tag. This can also return multiple elements which match.
- `querySelectorAll`: This is used to select elements by giving a valid CSS selector which identifies an element. It returns all the elements which satisfy the selector condition.

Getting and updating element content

Once we get access to the node or element by direct selection or through DOM traversal, the next set of methods enable you to get the content and make updates to the selected part:

- `nodeValue`: This is used to get the text content of a text node:
```
let nodeValue =
document.getElementById('refNode').firstChild.nodeValue
```
- textContent: This is used to get or set the text of a containing element:

```
document.getElementById('id').firstChild.textContent
= "Id text";
```

Adding and removing HTML content

Using the following methods you can modify the html content of the page by changing the text, adding or removing html elements as listed:

- `innerHTML`: By using this property you can get or set HTML content. Set will replace the content with the new content:
  ```
  document.getElementById('myId').innerHTML
  =‘<ul><li>Angular</li><li>Vue</li><li>React</li></
  ul>’;
  ```
  If you run the code, you will notice that everything inside the `div` with the ID of myId will be replaced with the new list.

  If we don't want to replace the content we can use `createElement()`, `createTextNode()`, and `appendChild()` to make changes to the HTML content.
- `createElement`: It is used to create a new HTML element as shown:
  ```
  let newElement = document.createElement('span');
  ```
- `createTextNode`: It is used to create a text node as shown:
  ```
  let text = document.createTextNode('Text Added!');
  ```
- `appendChild`: It is used to append a new element into a parent element.

  The four steps would include the preceding three methods to change content create a new element, .create a text node to be rendered and append it to the element created, finally append it to the location/ element where you want to place the new text:
  ```
  let newElement = document.createElement('span');
  let text = document.createTextNode('Text Added!');

  newElement.appendChild(text);
  document.getElementById('division').appendChild(newEle
  ```
- `removeChild`: It is used to remove HTML elements. It is called on the parent node of the element which needs to be removed.

  First get the element to remove, and then get its parent node. Then call the method `removeChild` to remove the element from the `parentNode` as shown in the following code:
  ```
  let toBeRemoved = document.getElementById('head');
  let parent = toBeRemoved.parentNode;
  parent.removeChild(toBeRemoved);
  ```

Attribute related

Next, you will learn how to manipulate the attributes of the DOM elements using the following mentioned methods:

- `getAttribute`: It is used to get an attribute:
```
let anchor1= document.getElementById('myId');
let srcURL= anchor1.getAttribute('href');//get href
for anchor tag
```
- `setAttribute`: It is used to set a new attribute to an element. It takes two arguments, first the attribute and second the value to be set for the attribute:
```
anchor1.setAttribute('href','www.google.com');
```
- `hasAttribute`: It is used to check if an attribute exists on an element:
```
var chkFlag= anchor1.hasAttribute('title');
```
- `removeAttribute`: It is used to remove an attribute; it takes an attribute as an argument:
```
anchor1.removeAttribute('title');
```

Using the above methods, you can access and make changes to the dom and the content of your webpage through JavaScript code.

Conclusion

At the end of this chapter, you should have a basic understanding of the JavaScript syntax, how to work with variables, arrays, functions, and many more, and how it can be used to work with the actual DOM to make changes to it. In the next chapter, you will be introduced to the Functional programming aspect of JavaScript.

Questions

1. The ___ method is used to create a text node.

   A. createElement

   B. createTextNode

   C. appendText

   D. None of the above

   Answer: B

   createTextNode is used to create a text node when it can be appended to any element using appendElement.

2. ____ changes the array by specifying the start position and number of elements to be replaced/removed and the new list used for replacing.

A. splice
B. slice
C. map
D. push

Answer: A.

3. This block scoped variable declaration is used whenever the containing value is constant and will not undergo change.

A. constant
B. const
C. var
D. let

Answer: B.

4. The function defined using expression gets created at runtime.

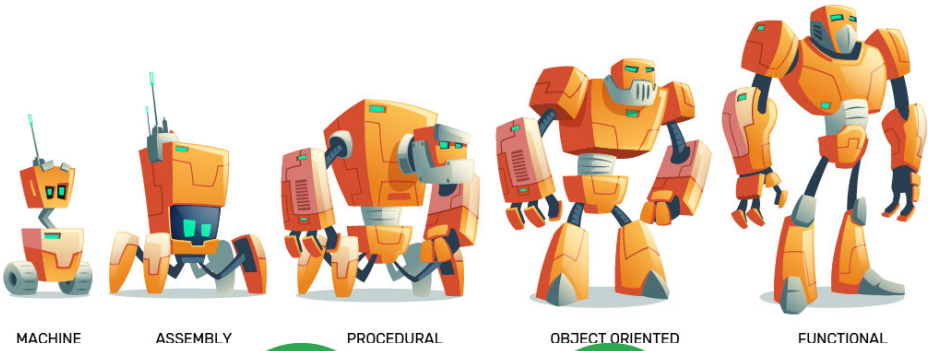
A. TRUE
B. FALSE

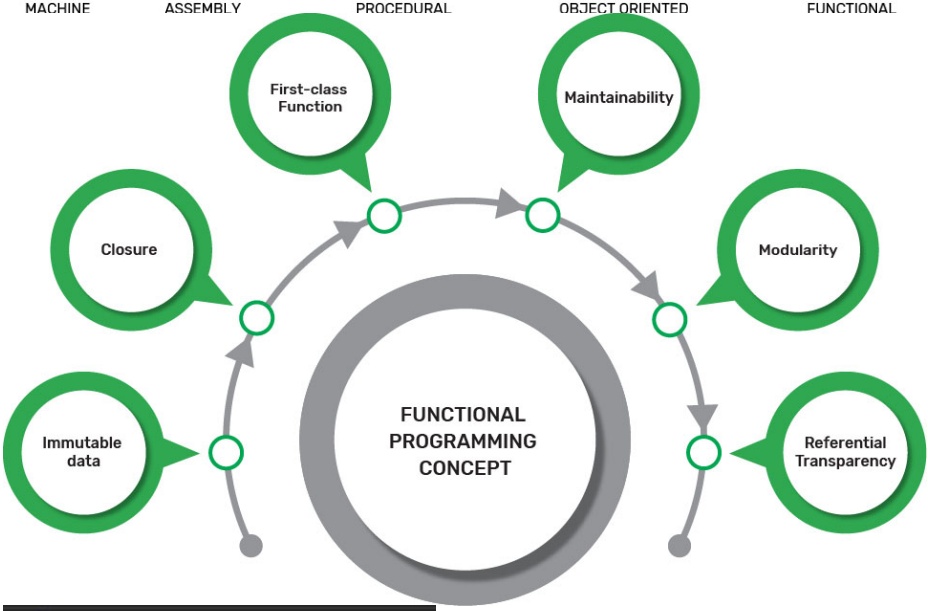
Answer: A.

5. This is a complex data type for handling complex data structures, including data of different types.

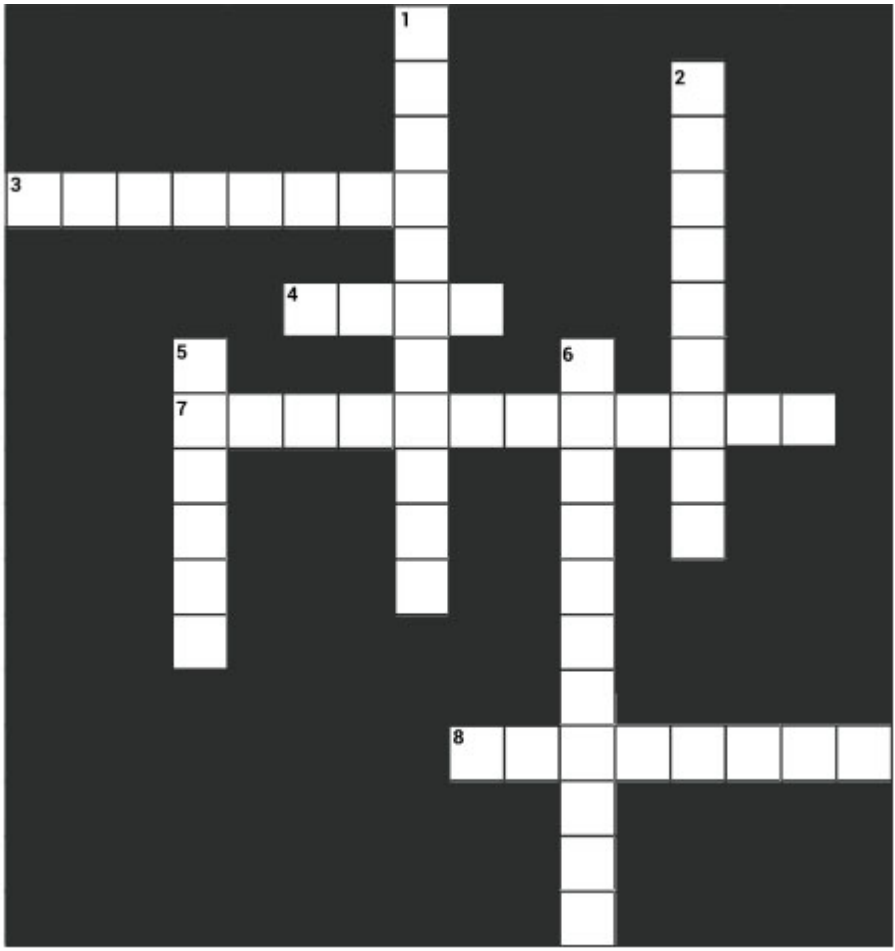
A. Array
B. Enum
C. Object
D. Symbol

Answer: C.


MACHINE
ASSEMBLY
PROCEDURAL
OBJECT ORIENTED
FUNCTIONAL


First-class Function
Maintainability
Closure
Modularity
Immutable data
FUNCTIONAL PROGRAMMING CONCEPT
Referential Transparency

```
4
3
2
1
Stop
```



HORIZONTAL

3. Functional programming will be used when you want to do multiple things_?_

4. Functional will not have observable side effects

7. Ensures that it won't change orginal state

8. Any function that is passed as an argument is called a _?_ function

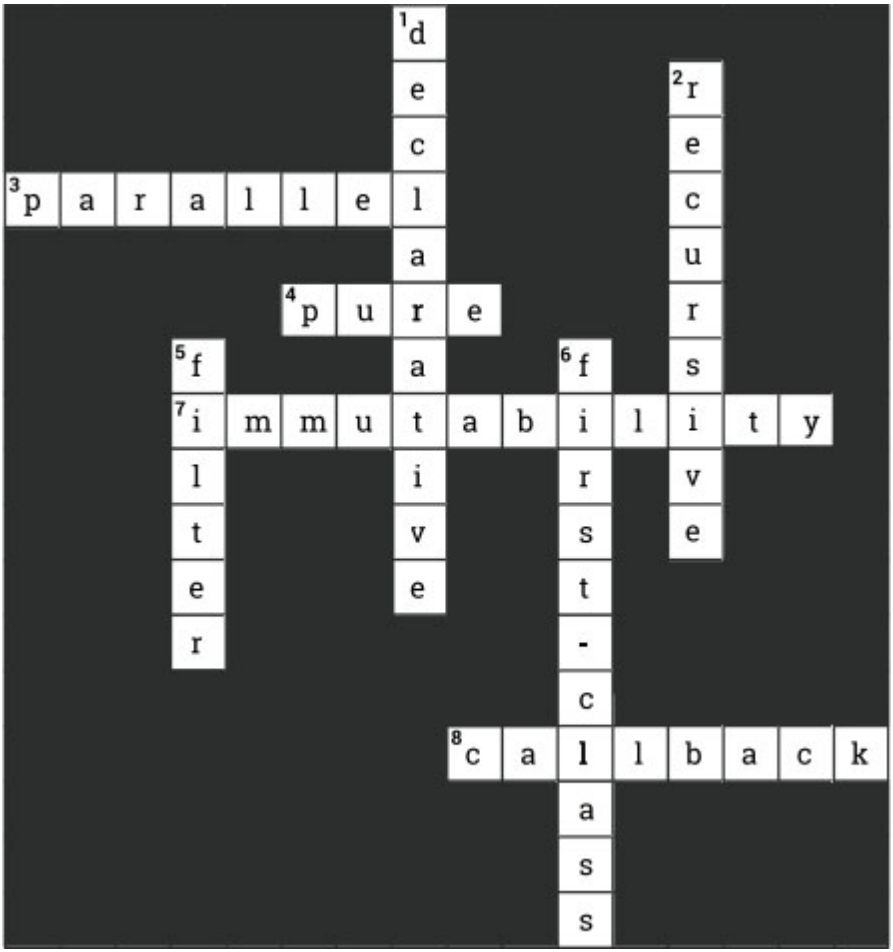
VERTICAL

1. Programming paradigm which focuses on what needs to be achieved instead of instructing how to achieve it.

2. Functional repeatedly call themselves until it reaches the base case

5. The _?_ method creates a new array with all elements that pass the test implemented by the provided function.

6. ? functions when functions in that are treated like any other variable.



CHAPTER 5

Functional Programming with JavaScript

"Object-oriented programming makes code understandable by encapsulating moving parts. Functional programming makes code understandable by minimizing moving parts."

~ Michael Feathers, author of 'Working with Legacy Code' (via Twitter)

Figure 5.1

One of the popular jargons in the web industry is functional programming. **Functional programming (FP)** is a highly admired approach to writing

code, and its popularity in developing commercial software is increasing day by day.

We will discuss what exactly functional programming means in more detail, but first let's get some basic idea about it. As enterprise applications (programs) got bigger, it became more complex and difficult to maintain, and in order to add new capabilities to the system, additional plugins or programs were added and the application worked. This application works, but if any issue comes up, it's very difficult to understand and fix it. As a programmer, working on something you do not really understand is no less than a nightmare. We should always try to keep the complexity of our programs as low as possible. An important way to do this is to try and make the code more abstract.

This chapter covers the fundamental concepts of functional programming which help you decide when and why to choose FP to apply in your day-to-day coding activities and how to relate it to JavaScript.

Structure

- Functional programming
- Features of functional programming
- When to use functional programming
- Major concepts of functional programming
- Limitations of functional programming language
- Conclusion

Objective

At the end of this chapter, you should be able to use a functional pattern with JavaScript in order to replace the common procedural programming pattern. Also, you can do this more with less and clean code.

Functional programming

Functional programming (also called FP) is a programming paradigm that defines the way a programmer thinks for solving a problem. FP is a pattern for developing programs using pure functions, which avoid the shared state and mutable data.

FP is not a language. It is language independent. It’s rooted conceptually in mathematics and the key principle is that all computations are execution of a series of mathematical functions.

Functional programming is a form of declarative programming. Let’s first understand what declarative programming is.

Knowingly or unknowingly as a web programmer/engineer, you are already using both the paradigms as part of your code, that is, the imperative and declarative syntax.

"You know imperative programming is like how you do something, and declarative programming is more like what you do or something. "~ from a blog by Tyler McGinnis

Let's understand this using some code in imperative style, focusing on the how part.

Imperative style

The following example shows how to filter out the even numbers from a given array of numbers in the imperative style:

```
constmyArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
function fetchEven(myArray){
  let evenNos = [];
for(let i = 0; i<myArray.length + 1; i++){
       if ( i % 2 == 0 &&i != 0 ){
evenNos.push( i )
       };
      };
 return evenNos
 };
console.log(fetchEven(myArray)); // logs [2, 4, 6, 8,
10]
```

Declarative style/functional

Now, let's do the same thing in a declarative way and using a different variable to avoid any error:

```
constmyDArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
function fetchEvenD(myArray){
  return myArray.filter(num =>num % 2 == 0)
}

console.log(fetchEvenD(myDArray)) // logs [2, 4, 6, 8,
10]
```

From the preceding example, the declarative/functional approach produces shorter code, and it's much easier to understand by a programmer, reflects clean code styles, and also helps in debugging and testing.

Let's understand what's really happening in the preceding example.

In the imperative style of coding, we are writing every step in order to

retrieve odd numbers from the existing array. We provided an array to the function; the function starts with creating an empty array which is used for storing even numbers. We iterate over the input array and check whether the provided number is even and push it to the even array. We also added an extra condition to check whether the number is zero or not to avoid pushing it.

In the functional/declarative style of coding, we define the result we are expecting using a filter method and allow the system to take care of all the other steps. This is a more declarative approach.

Now, revisit our definition which talks about the programming style using declarations and expressions rather than statements.

Following are few programming languages that support functional programming:

- Haskell
- JavaScript
- Scala
- Erlang
- Lisp
- Clojure
- Common Lisp

Features of functional programming

Functional programming languages have the following features:

- **No state:** Programs do not have state, that is, data in the FP language is immutable and functions cannot change state.
- **Order of execution insignificant:** As all functions work independent of other, the order of function executions in FP does not have great significance and still leads to the same results.
- **Modularity:** FP emphasizes on writing smaller and independent units called pure functions to support immutable nature. This means FP supports better modular programming than OOP.
- **First class citizens:** Functions: Functions are first class objects and they are small independent units and hence can be executed in any order and generate the same results.
- **Higher-order functions and lazy evaluation:** Functional programming supports higher-order functions and lazy evaluation features. (Don't worry we will soon see what this means).

Using functional programming

We should think of functional programming for the following scenarios:

- When you need results instead of how things are being done
- When you want to do multiple things in parallel
- A lot of operations/activities need to be performed on a fixed set of data

Major concepts of functional programming

The following are important concepts of functional programming:

Figure 5.2

Pure functions

To understand functional programming, the very basic and first concept you need to learn is pure functions. Let's understand what this means in detail.

To qualify a function as pure, it must have the following characteristics:

- For the same set of arguments provided, the function returns the same results always.
- The pure function will not have observable side effects.

Example: Math.sign(x): For x value, we can determine what can be the result and it won't change the value of x. Also, it does not have any side effects like making any network calls or logging, and so on.

We can state that any function which does activities other than just returning a value can be considered as impure.

Let's understand the preceding points with the example code.

For the same set of arguments, the provided function returns the same results always.

Problem: Write a function to calculate the area of a sphere:

```
Area of sphere = 4Π r2
let PI = 3.14;
constcalculateAreaOfSphere = (radiusValue) => 4 * PI *
radiusValue * radiusValue;
calculateAreaOfSphere(10); // returns 1256
```

The preceding function `calculateAreaOfSphere` is not a pure function because it accesses the global state value, that is, PI value. If the PI value is changed, which is outside the scope of the function, it will impact the outcome of the function.

That is, if `PI = 3.142857142857143`, then the `calculateAreaOfSphere` function returns a different result:

```
let PI = 3.142857142857143;
constcalculateAreaOfSphere = (radiusValue, pi) => 4 *
pi * radiusValue * radiusValue;
calculateAreaOfSphere(10,PI); // returns
1257.142857142857
```

In the preceding function, we are passing the PI value as one of arguments and hence we have no access to the global state and for the same set of input parameters, the `calculateAreaOfSphere` function will give the same results.

A pure function will not have observable side effects:

To understand the function side effects, we need to include manipulating a global object or a parameter passed by reference.

Let's understand this with some code, and write a function to count how many times a button has been clicked:

```
let buttonClickCount = 0;
function buttonClick(value) {
buttonClickCount = value + 1;
}
buttonClick(buttonClickCount);
console.log(buttonClickCount); // 1
```

We have the `buttonClickCount` value. The preceding function is an impure function. It receives the click count and it re-assigns it to the value by incrementing by 1; hence, we are violating the mutability feature of FP.

So, we rewrite the preceding example using the pure function:

```
let buttonClickCount = 0;
constbuttonClick = (value) => value + 1
buttonClick(buttonClickCount);
console.log(buttonClickCount); // 1
```

In the preceding example, we did not modify the global state and hence there are no side effects.

Benefits of pure functions

The following are key benefits of pure functions:

- **Testability:** As for the same input, the output is the same. Writing a unit test becomes too easy and you don't need to mock any data.
- **Reusability:** Using utility functions and a component-based approach

of pure functions, we can reuse a lot of code/functions.

- **Maintainability:** A function with no side effects and which does not impact any other piece of code is easy to maintain.

Higher-order functions

Higher-order functions are functions:

- Which take other functions as arguments
- Returns another function as part of the result

Functions are nothing but values in JavaScript so we can pass them as an argument to another function. Higher-order functions are mainly used for compositions.

You might have already heard about filter, reduce, and map functions; these are some of the examples from higher-order functions.

Let's understand this with some code.

Declare array of objects:

```
const cities = [
{ city: 'Newyork', country: 'USA' },
{ city: 'Mumbai', country: 'India' },
{ city: 'Shanghai', country: 'China' },
{ city: 'New Delhi', country: 'India' },
{ city: 'Tokyo', country: 'Japan' },
{ city: 'Bengaluru', country: 'India' }
 ];
```

Now, if you want to filter Indian cities from an array, then this can be done using the imperative paradigm, which is, using for loop:

```
constindianCities = [];
for (let i = 0; i<cities.length; i++) {
 if (cities[i].country === 'India') {
indianCities.push(cities[i]);
 }
}
```

Let's understand what's happening. We are iterating through every record of the city and whenever the country belongs to India, we are pushing the record/city into our result array.

Now, see how we can do this using filter:

```
constindianCities = cities.filter(function(entry) {
   return entry.country === 'India';
 });
```

In the preceding example, `filter` returns new set of arrays created from all entries that passes condition of country belongs to India or in general any kind of condition. It provides a callback function for each entry.

From the preceding example, you saw that it helps in composition and we can also reuse it. Filtering of cities is taken care natively by abstracting logic from the user. It looks like clean coding style isn't it?

Immutability

In simple words, mutability means liable or subject to change or alteration. In the context of programming, it means the object's state is allowed to undergo change, whereas immutability is exactly the opposite of this. That is, once an object is created, it can never change its state.

Immutability ensures that it won't change its original state, that is, the value of an object should return the new copy of state or value after processing.

For developers, it's very tough to debug if something goes wrong in cases where objects are mutable.

In functional programming, a developer can't modify variables once they have been initialized. This helps developers throughout the lifespan of the program as he can be confident that no one is going to change the value of the object/variable.

All primitive types in JavaScript are immutable in nature. For example, `Strings, Numbers`, and so on. Let's see some of the examples:

```
constsimpleString = "Hello world!";
constotherString = simpleString.slice(6,12);
console.log(simpleString); // Hello world!
console.log(otherString); // world!
```

In the preceding example, we declared the string and applied mutable function on it, but it doesn't change the original value of the variable but it returns a new copy of the original value.

Also, JavaScript by default does not provide anything to make the object and array immutable. Let's understand this by an example for arrays:

```
varmyArray = [1,2,3,4,5];
myArray.push(6); // This statement mutates array and
add value to it
console.log(myArray); // [1,2,3,4,5,6]
```

In the preceding example, the array i mutated using the push method (other mutation methods are `pop(), splice(), sort(),` and so on).

So, the question is how do we make arrays and objects immutable?

Well, there are non-mutating methods like `map()`, `filter()`, and so on or using existing libraries as follows:

- Immutable JS
- Seamless-immutable

Recursion

In a functional programming language, there are no loops, that is, `for`, `while`. So, the question is how do we iterate? Recursion solves this problem. Recursive functions repeatedly call themselves until they reach the base case.

A program in which a function calls itself is called **recursion** and the related function is known as a recursive function as shown in the following example of Fibonacci series:

```
fib(n){
if (n <= 1)
 return 1;
else
 return fib(n - 1) + fib(n - 2);
}
```

Recursion solves the problem by splitting or representing it in smaller problems, and also adds one or more base conditions to stop the execution or recursive function.

To compute the factorial of *n* (for example, *3! = 3 X 2 X 1 = 6),* we must know the factorial of (*n-1*); here, the base case for this is *n=0*. Return 1 if *n=0:*

```
intfact(int n) {
    if (n < = 1) // base case
        return 1;
 else
        return n*fact(n-1);
 }
```

Here is a simple example to understand the concept with stopwatch behavior:

```
let countdownTime = (count) => {
 if (count === 0) { // base case
      console.log("Stop");
 }
 else {
      console.log(count);
countdownTime(count-1); //recursive call on a reduced
```

```
problem countdownTime-1
  }
}
countdownTime(4);
O/P :
```

Figure 5.3

Hence, recursion suits best when you need to call the same function repeatedly with different parameters.

Another important point to be considered is that it is favored in FP because it doesn't require setting and maintaining the state with local variables.

Recursive functions are pure in nature and they perform no side effects on variable states; hence, easy to test.

Referential transparency

Expression is called **referentially transparent** if it can be replaced with its corresponding value without changing the program's behavior.

In short,

Pure functions + Immutable data = Referential Transparency

Advantages of referential transparency are as follows:

- It eliminates side effects of code.
- Makes your code context-independent, which means your code can run in any order and any context; it will always return the same results.

Let's understand this with an example.

Suppose if we have the sum function:

```
const sum = (x, y) => x + y;
sum(6, sum(10, 20)); // calling with 5 & 8 as
parameters
```

In the preceding statement, `sum(10,20)` always equals to 30, so we can very well replace it with 30, that is, `sum(6,30)`.

Similarly, `sum(6,30)` results into 36; hence, we can replace the expression with a constant value, that is, 36 and optimize it.

First-class functions

The concept of functions as first-class entities is that functions are also treated as values and used as any other data.

Functions can be:

- Stored in a variable
- Passed as an argument to any other function
- Returned by any other function

Stored in a variable

Functions can be stored in three different ways:

- Variables
- Objects
- Array

Storing function in a variable is shown in the following example:

```
const foo = function() {
console.log("Hello world!");
 }
// Invoke it using the variable
foo();
```

In the preceding example, an anonymous function is assigned to a variable and the same variable is used to call that function by adding parentheses () at the end.

The preceding example can be re-written by giving a name to the function:

```
const foo = function greetWorld() {
console.log("Hello world!");
 }
// Invoke it using the variable
foo();
```

Storing function in an object is shown in the next example:

```
let obj = {
greetWorld : function(){
console.log("Hello world!");
    }
 }
// Invoke
obj.greetWorld(); // Hello world!
```

Storing function in an array as shown in the next example:

```
let arr = [ function greetWorld(){ console.log("Hello
world!"); }];
// Invoke
arr[0](); // Hello world!
```

Passed as an argument to any other function

In the preceding example, you saw how we are treating a function as a value and `greetWorld` is the `callback` function.

Callback is a function passed as an argument to another function.

Returned by any other function

We can rewrite the preceding example to return a function because we are treating the function as a value:

```
function greet(){
 return function (){
console.log('Hi Ranjit!');
    }
 }
greet()(); // Hi Ranjit!
```

In the preceding example, the first parentheses will return the function and the second one will invoke the returned function.

Limitations of a functional programming language

Though functional programming languages have many benefits, it has few limitations or drawbacks like:

- As FP is stateless and creates new objects every time which in turn results into use of more memory
- FP is not easy to understand for beginners
- Objects may not represent the problem correctly
- Integrating pure functions into a program is a bit difficult

Conclusion

In this chapter, we learned that the functional programming paradigm focuses on results, not the process. It helps programmers in writing better code in terms of reusability, modularity, performance, extensibility, and testability. In the next chapter, you will learn about another important programming methodology—Object-oriented Programming.

Questions

Complete the following crossword to refresh your knowledge:

Figure 5.4

Figure 5.5

Answer:

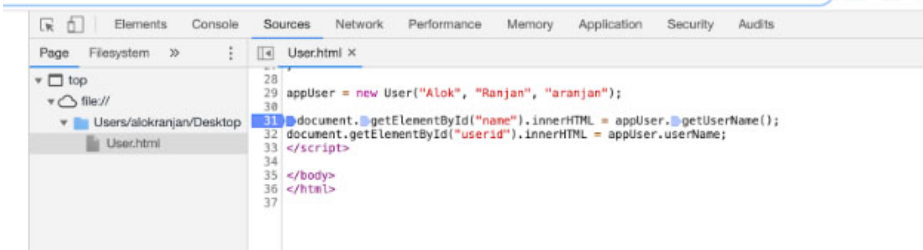
Figure 5.6

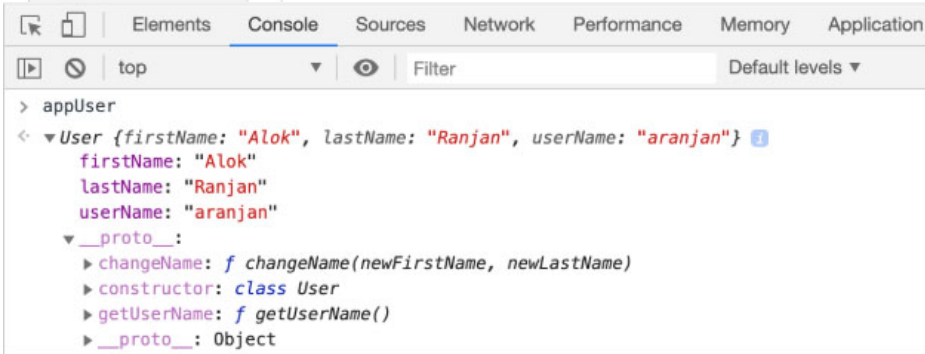
<Class Name>

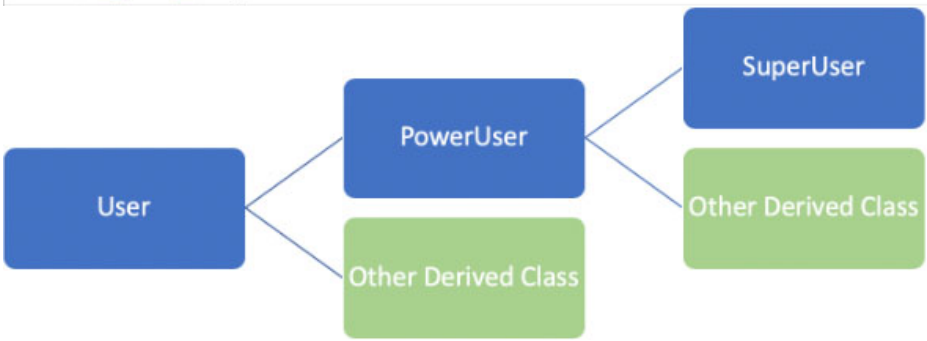
<Fields / States>

<Methods for working with the states/fields>

```
User Count = 1
Abhilasha Sinha
User Count = 2
Ranjit Battewad
```







CHAPTER 6

Object-Oriented JavaScript

"OOP to me, means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things."

~ ALAN KAY

When Alan Key coined the term **Object-Oriented Programming (OOP)** in 1966-1967, he envisioned that instead of data being passed here and there procedurally, the computers should talk like real objects—through messaging. Like in the real world you come across the need for local retention and protection and hiding of states—the computer program shall also allow that. That is what formed the base for early OOP languages like SmallTalk and hundreds of OOP-based languages have been developed since then. It was not at all a surprise that JavaScript being versatile had left scope for procedural as well as Object-Oriented paradigm. With ECMAScript 2015 formally introducing the concepts like class, it has spiced up the usage of OOP in JavaScript, wherein people were making use of a slightly more complicated approach to achieve the same.

This chapter will introduce you to the various concepts of (OOP using the latest ECMAScript syntax of JavaScript. You will start with gaining an understanding of what classes and objects are, followed by the application of OOP concepts like encapsulation, inheritance and abstraction.

Structure

- Introduction to OOP
- Class
- Object
- Encapsulation
- Inheritance
- Abstraction

Objective

After studying this chapter:

- You will learn the basics of **Object-Oriented Programming (OOP)** and in specific get introduced to the OOP concepts using JavaScript.

- Later in this book, we talk about React.js as a UI framework; the concepts learned in this chapter will be applicable across all the JavaScript frameworks.

Introduction to OOP

Object-Oriented Programming (OOP) is a programming paradigm wherein the emphasis is on ensuring that the objects in question should know what they can or cannot do. The objects interact with other objects through the allowed interfaces (the public fields and methods), and they respect the constraints applied through the various OOP concepts.

At a very high level, an OOP paradigm defines a class, and you make use of that class to create an instance, which is called the object of that class. The following diagram shows a very basic structure of a class:

Figure 6.1: *Class structure*

As depicted in the figure, a class is associated with a name, has some properties(fields/state) and some methods, which work with the data elements/properties.

Class

Before ES6, developers were implementing class explicitly using JavaScript functions, but with ECMAScript 2015, the class keyword was formally introduced. This saves certain coding effort and provides better motivation to appreciate and apply the OOP concepts in JavaScript. While you can make it sound trivial that it is just syntactic sugar in the form of a predefined function, however, treating this as a function which can change the behavior of the coder, let's call it a very special function and you shall use it whenever you are in need of defining an entity to create one or more objects of that type.

At the simplest level, a `class` looks like as shown as follows:

```
class User {
constructor( firstName, lastName, userName) {
this.firstName = firstName;
this.lastName = lastName;
this.userName = userName;
 }

getUserName() {
  return this.firstName + " " + this.lastName;
 }

changeName(newFirstName, newLastName) {
```

```
this.firstName = newFirstName;
this.lastName = newLastName;
 }
}
```

Here you just defined a class with the `firstName`, `lastName`, and `userName` as member variables and you also defined `getUserName` and `changeName` as methods, which can be used to read/manage the state of the class.

In the methods, you use the `this` keyword to refer to the members of the class like `this.firstName`, `this.lastName`. It gives access to the current instance of the member.

Once you have the class defined, you are all set to create as many instances of the class as needed in your application.

Further, a class expects you to write all the code in the strict mode. In JavaScript, the strict mode is a preventive way of writing code to impose good coding practices, and it throws errors when relatively *unsafe* actions are taken (for example, when you try to use undeclared variables).

The constructor of a class

Every class must have a constructor. You can either declare it explicitly as shown in the above example or you can skip explicit definition, in which case the JavaScript will automatically add an empty and invisible constructor.

When you define a constructor explicitly, it can have zero or more parameters, which can be used to initialize the member variables. In the preceding class declaration, you have used an explicit class declaration.

Also, there are cases where a default initialization will be quite okay. You can assign a default value by assigning a value to the parameters of the constructor as shown as follows:

```
class User {
constructor(firstName, lastName, userId = "asinha") {
this.firstName = firstName;
this.lastName = lastName;
this.userId = userId;
 }

getUserName() {
  return this.firstName + " " + this.lastName;
 }

changeName(newFirstName, newLastName) {
this.firstName = newFirstName;
```

```
this.lastName = newLastName;
 }
}

appUser = new User("Abhilasha", "Sinha");
console.log( appUser.getUserName());
```

In the preceding code, the `constructor` passes just two parameters, and it uses the default value of the `userId` to initialize the corresponding value.

Static member in a class

A static member of a class implies a member which is associated at the class level and not at the object instance level. A static member has only a single instance at the class level.

Earlier static functions were defined by adding a method / variable on top of a given function object. However, with ES6, JavaScript provides you with an approach that you would expect in any OOP paradigm. The `static` keyword can now be used to indicate a given method as a static method.

For example, in the following code, you can make use of the create method, which is a `static` method to create an instance of the `User`:

```
class User {
 static userCount = 0;
constructor(firstName, lastName, userId = "asinha") {
this.firstName = firstName;
this.lastName = lastName;
this.userId = userId;
User.userCount++;
console.log('User Count = ' + User.userCount);
 }
getUserName() {
  return this.firstName + " " + this.lastName;
 }
changeName(newFirstName, newLastName) {
this.firstName = newFirstName;
this.lastName = newLastName;
 }
 static create(firstName, lastName, userId) {
  return new User(firstName, lastName, userId);
 }
}
appUser = User.create("Abhilasha", "Sinha");
```

```
console.log( appUser.getUserName());
anotherUser = User.create("Ranjit", "Battewad",
"rbattewad");
console.log( anotherUser.getUserName());
```

The following image shows how the output of the preceding code looks like in the console:

Figure 6.2: *Output of static member example*

With the concept of class in place, the static behaves more like what you will expect in the Object-Oriented Paradigm. You call static methods or variables on the class rather than on the instances. Typically, when you have a need for utility function or lifecycle functions like create, then a static method makes a lot of sense. In the preceding code example, you have used the static method to create a new instance (`anotherObject`) from the class. Also, you used the static member variable `userCount` to keep track of the number of instances of the `User` class.

Object

Whenever you need an instance of a class, you make use of the new operator in the class to create a new object. The new instance creation automatically calls the constructor of the class to initialize the member variables. For example, when you create an object `appUser` as shown as follows:

```
appUser = new User("Alok", "Ranjan", "aranjan");
```

The constructor of the User class will be automatically invoked with the parameters passed to the new operator. On the other hand, if you try to create an object without a new operator, then it results into exception. Further, other methods lack an internal Construct method, and hence you cannot call a new operator on them.

The following code shows a typical usage of the classes in a website:

```
<!DOCTYPE html>
<html>
<body>
<h2>User Details</h2>
<p id="name"></p>
<p id="userid"></p>

<script>
class User {
constructor( firstName, lastName, userName) {
this.firstName = firstName;
```

```
this.lastName = lastName;
this.userName = userName;
 }

getUserName() {
  return this.firstName + " " + this.lastName;
 }

changeName(newFirstName, newLastName) {
this.firstName = newFirstName;
this.lastName = newLastName;
 }
}

appUser = new User("Alok", "Ranjan", "aranjan");

document.getElementById("name").innerHTML =
appUser.getUserName();
document.getElementById("userid").innerHTML =
appUser.userName;
</script>

</body>
</html>
```

You can inspect the element and put a breakpoint after the creation of the new object:

Figure 6.3: *Inspecting the element from Chrome dev tools*

On the console you can notice that the `appUser` is like any other object in the JSON format and additionally it also shows the methods available for this object, which it acquired from the class:

Figure 6.4: *Logging the object value in the console*

Note:

As you would have guessed that constructor, like any other function, is available for the call, the only difference being that the constructor requires the usage of the new operator. Although, technically you can create an object from another object as shown:

```
oneMoreObject = new appUser.constructor("Abhilasha",
"Sinha", "asinha");
```

In order to keep the behavior consistent, it is best that you call a new operator on the class name.

Class expression

Earlier in this chapter, you learned about declaring and using class to create an object of a class. That is the approach you shall use in general. However, there are situations wherein you can declare a class as an expression. For example, you may have a need to return a class from a function dynamically or you may need to assign an anonymous class declaration to a variable. The following code snippet shows how you can create a class anonymously and assign it to a variable, which can be later used to create instances of the class:

```
<script>

let AppUser = class {
constructor( firstName, lastName, userId) {

this.firstName = firstName;
this.lastName = lastName;
this.userId = userId;
 }

getUserName() {
   return this.firstName + " " + this.lastName;
 }

changeName(newFirstName, newLastName) {
this.firstName = newFirstName;
this.lastName = newLastName;
 }

whoAmI() {
   return "I am a User!";
 }
}

appUser = new AppUser("Alok", "Ranjan", "aranjan");

console.log( appUser.whoAmI() );
</script>
```

Since classes can be declared as an expression, it also highlights that eventually creating a class in JavaScript is like evaluating an expression. While the preceding code shows an anonymous way of declaring a class, you can also provide a class name and use that inside your class.

Encapsulation

Encapsulation is the mechanism through which a class wraps the data and methods, into a single unit and provides a mechanism for data hiding, by

declaring the data as private and providing public methods to access and update the data. By default, a variable is declared as public. However, you can use the hash (#) prefixed variable names, where the hash is the part of the name, which will be used as a private variable and it will be accessible only from inside the class.

The following code shows the usage of the public and private variables, including the static private and public variables:

```
class User {
 static userCount = 0;
 static #passwordLevel = 1;
phoneNumber = "+91 2993849384";
 #email = "abhilasha.sinha@walkingtree.tech";

constructor(firstName, lastName, emailId = "", userId =
"asinha") {
this.firstName = firstName;
this.lastName = lastName;
this.userId = userId;
this.#email = emailId;

User.userCount++;

console.log('User Count = ' + User.userCount);
 }
getUserName() {
  return this.firstName + " " + this.lastName;
 }

changeName(newFirstName, newLastName) {
this.firstName = newFirstName;
this.lastName = newLastName;
 }
 static create(firstName, lastName, emailId, userId) {
  return new User(firstName, lastName, emailId,
  userId);
 }
whoAmI() {
  return "I am a User! " + " My email ID is " +
  this.#email + " and my Password Level = " +
  User.#passwordLevel;
 }
}
appUser = new User("Abhilasha", "Sinha");

console.log( appUser.getUserName() + " with the Phone
Number: " + appUser.phoneNumber);
```

```
anotherUser = User.create("Ranjit", "Battewad",
"ranjit.battewad@walkingtree.tech", "rbattewad");

// console.log( anotherUser.getUserName() + " has a
password level = " + User.#passwordLevel); <-- This
will error out as you are trying to access private
static variable.
console.log( anotherUser.whoAmI());
```

The highlighted code in the preceding code snippet indicates the following:

- You can define public and private members
- The private members, including the private static member (for example, `passwordLevel`) are accessible only from inside the class
- Public variables can be accessed by creating an instance of the class
- The public static member variables will not be called on a specific instance; instead, they can be called by directly using the class name

Similar to the member variable name, you can append hash before the method name to declare a method as a private method of the class.

Inheritance

Inheritance is a mechanism in OOP, which allows you to base your class on an existing class and thus to acquire the properties and methods exposed by the existing class. This improves reusability and saves effort and cost during the development as well as maintenance of the application. While previously, JavaScript had a complicated way of achieving inheritance, now the extends keyword can be used to indicate that a class is derived from a given base class.

The following code snippet demonstrates inheritance using the extends keyword to create a derived class:

```
class User {
constructor( firstName, lastName, userId) {
this.firstName = firstName;
this.lastName = lastName;
this.userId = userId;
 }
getUserName() {
  return this.firstName + " " + this.lastName;
 }

changeName(newFirstName, newLastName) {
this.firstName = newFirstName;
this.lastName = newLastName;
```

```
 }
}

class PowerUser extends User {
constructor( firstName, lastName, userId, role ) {
super(firstName, lastName, userId);
this.role = role;
 }

get userRole() {
  return this.role;
 }
 set userRole( newRole ) {
this.role = newRole;
 }
}
```

As shown in the following code, you can create an object exactly the way you created earlier and use methods of the derived class to create the HTML element:

```
appUser = new PowerUser("Alok", "Ranjan", "aranjan",
"Manager");
document.getElementById("name").innerHTML =
appUser.getUserName();
document.getElementById("userid").innerHTML =
appUser.userId;
document.getElementById("roleId").innerHTML =
appUser.userRole;
```

The `appUser` created previously can be considered as an instance of the derived class as well as the base class, and the following console message shall return true in both the cases:

```
console.log( appUserinstanceofPowerUser);
console.log( appUserinstanceof User);
```

Inheritance can be used to reuse base class definitions and build derived specialized classes over it.

Use of super inside the constructor

The derived class expects you to use the super keyword to be able to invoke the constructor of the parent class. This indeed makes great sense because you don't want to repeat the initiation logic of parent classes into the derived class. Especially when your class has multiple layers, as shown in the following diagram, it appears to be a great savior:

Figure 6.5: Inheritance

In case you don't explicitly define a constructor in the derived class, then JavaScript automatically defines a constructor for you, and it automatically calls the super keyword which invokes the constructor method of the parent class.

However, if you explicitly define a `constructor` in the derived class, then it must invoke the parent constructor by explicitly calling super in the derived class before accessing `this` or returning from a derived constructor. Since the `super` call will initialize the `this` object, even within the constructor, it is important that you call super before you use this to initialize any other member variable.

Once you have a well-defined inheritance in place, you can check the derivation of the class from a given base class using the `instanceof` operator as shown in the following snippet:

```
console.log( appUserinstanceofPowerUser);
console.log( appUserinstanceof User);
```

The preceding code shall return true for both the statements, that is, for a base class as well as derived class.

Method override

By default, the methods of the base class will become available to the derived class with the same behavior. However, there are situations where you may want to change the characteristics or behavior of some of the methods which were derived from the parent or base class. This process is called a **method override**.

When you override a method in the derived class and invoke the overridden method, the method defined in the most derived class (using which you have created the object) will be executed. Of course, that is what you would expect. In the following example, you will see a new method, `whoAmI` being added in the User as well as `PowerUser` class and the one corresponding to the `PowerUser` will be invoked:

```
class User {
constructor( firstName, lastName, userId) {
this.firstName = firstName;
this.lastName = lastName;
this.userId = userId;
 }
getUserName() {
   return this.firstName + " " + this.lastName;
 }
changeName(newFirstName, newLastName) {
```

```
this.firstName = newFirstName;
this.lastName = newLastName;
 }
whoAmI() {
  return "I am a User!";
 }
}

class PowerUser extends User {
constructor( firstName, lastName, userId, role ) {
super(firstName, lastName, userId);
this.role = role;
 }
 get userRole() {
  return this.role;
 }
 set userRole( newRole ) {
this.role = newRole;
 }
whoAmI() {
  return "I am a PowerUser!";
 }
}
appUser = new PowerUser("Alok", "Ranjan", "aranjan",
"Manager");
console.log( appUser.whoAmI() );
```

The execution of the preceding code results into the printing of "`I am a PowerUser!`" on the console.

Calling the parent function

Whenever you work with object-oriented concepts, there is always a question regarding how do we call the overridden function? In JavaScript, you still need to remember that class is more like syntactic sugar. Unlike other OOP languages like Java/C++, when you create an instance of a derived class, it doesn't create an instance of the parent or base class. Only one instance of the class is created depending on the object you used to create the instance. Hence, you cannot call the overridden function of the parent class from the created object.

However, if you still need a way of calling the overridden method of the parent class from the child class, then it can be done by using `super.methodName()` syntax, which will invoke the `overridden` method. For example, the following code will print output as if the `whoAmI` or child, as well as the parent, has been called:

```
class User {
 …
whoAmI() {
 return "I am a User!";
 }
}
class PowerUser extends User {
 …
whoAmI() {
console.log(super.whoAmI());
 return "I am a PowerUser!";
 }
}

appUser = new PowerUser("Alok", "Ranjan", "aranjan",
"Manager");
console.log(appUser.whoAmI());
```

As you can see in the code, the parent's overridden method can be referenced using the super keyword and achieve the requirement to be able to invoke the parent method from the child.

Inheriting from built-in objects

In JavaScript, you come across various in-built objects which helps you to manage various aspects of programming logic, for example, collections (indexed/keyed), fundamental objects, properties (functions/values), control abstractions, reflections, number, text, dates, and namespaces like Webassembly and Intl. You may like to refer to the following URL to know more about the built-in objects (also known as the **global objects**):

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects.

While the earlier version of JavaScript allowed you to make use of complicated and inconsistent way of inheriting properties of a built-in object (because essentially you were first creating a derived object and then decorating them with the properties of the object), the ES6 has made it simpler by allowing you to simply use class-based extend keyword where it first creates the base object and then applies the derived class properties and methods as per the derived class constructors.

In the following example, you will see a popular object, Map being inherited to create a CustomMap class:

```
class CustomMap extends Map {
 get(entry) {
  let valueForKey = super.get(entry);
```

```
  if ( valueForKey == undefined ) {
   return "The value for the " + entry + " does not
   exist!";
  } else {
   return valueForKey;
  }
 }
}

let userArray = [['rb', 'Ranjit Battewad'], ['as',
'Abhilasha Sinha'], ['ar', 'Alok Ranjan']];

let mapInstance = new CustomMap(userArray);

console.log(mapInstance.get('as'));

console.log(mapInstance.get('ps'));
```

When you execute this code, you can see the output as a value corresponding to the input key. Also, if for a given key, the value doesn't exist (for example, in the preceding code there is no entry for the key '`ps`') then the parent will return undefined. In this case, you have overridden the parent class method to output a more meaningful message.

Abstraction

In OOP, abstraction enables you to hide the unnecessary details from the user and thus expose only those details which are needed for them to be able to do their job.

Unlike languages like Java in JavaScript, we don't have direct support for abstraction. However, you can make use of `new.target` to check if a specific constructor has been called or not. Inside a constructor when you check for `new.target`, it refers to the constructor invoked by the new operator. Hence, if you don't intend to allow objects of any specific class to be created then inside the constructor of this class, you need to check for the value of `new.target` and if it matches the class name then throw an exception.

The following code shows that the User class will not be allowed to create an instance, while the `PowerUser` class can be used for creating an instance of the user:

```
class User {
constructor( firstName, lastName, userId) {
  if ( new.target === User ) {
   throw new TypeError("User is an abstract class. You
   cannot create an instance directly!");
  }
```

```
this.firstName = firstName;
this.lastName = lastName;
this.userId = userId;
 }

getUserName() {
 return this.firstName + " " + this.lastName;
 }

changeName(newFirstName, newLastName) {
this.firstName = newFirstName;
this.lastName = newLastName;
 }

whoAmI() {
 return "I am a User!";
 }
}

class PowerUser extends User {
constructor( firstName, lastName, userId, role ) {
super(firstName, lastName, userId);
this.role = role;
 }

get userRole() {
  return this.role;
 }
 set userRole( newRole ) {
this.role = newRole;
 }

whoAmI() {
  return "I am a PowerUser!";
 }

}

// abstractUser = new User("Alok", "Ranjan",
"aranjan"); <-- invoking this will throw an exception
from the constructor of the User class.

appUser = new PowerUser("Alok", "Ranjan", "aranjan",
"Manager");

console.log( appUser.whoAmI() );

</script>
```

While you can achieve a certain level of abstraction through this approach,

JavaScript still doesn't provide a full-fledged abstraction mechanism. Hence, from this perspective, the object orientation of JavaScript is still a work in progress.

Conclusion

Thinking in terms of objects and the interaction among the objects is always a great way to visualize and write logic. In this chapter, you learned about ways to make use of the new class system in JavaScript to declare and use classes and apply the object-oriented concepts like inheritance, override, abstraction, etc. In addition to creating your own classes, you also learned about extending built-in objects.

With this knowledge in place, now you are all set to think in terms of objects and write better-structured code.

In the next chapter, you will explore another aspect of JavaScript - Asynchronous programming. You will understand the meaning of the term Asynchronous from a programming perspective, the functioning of JavaScript Runtime Engine and the different JavaScript asynchronous constructs.

Questions

1. The ___ symbol is used to create a private member in JavaScript.

   A. $

   B. @

   C. #

   D. ~

   Answer: Option C

2. The method marked with the static keyword will have?

   A. An instance for each instance created from the class

   B. A single instance referred to using a new method

   C. A single instance referred to using the class name

   D. An instance for each object referred using the this keyword

   Answer: Option C. A static member is a single instance at the class level and accessed using the class name.

3. ___ is a mechanism in OOP, which allows you to base your class on an existing parent class and acquire the properties and methods exposed by the existing class.

   A. Inheritance

B. Encapsulation

C. Abstraction

D. Overriding

Answer: Option A.

4. The ___ keyword is used to access the parent class constructor from the extended class constructor.

A. new

B. extends

C. super

D. Parent

Answer: Option C.

5. Which of the following is true with respect to inheritance?

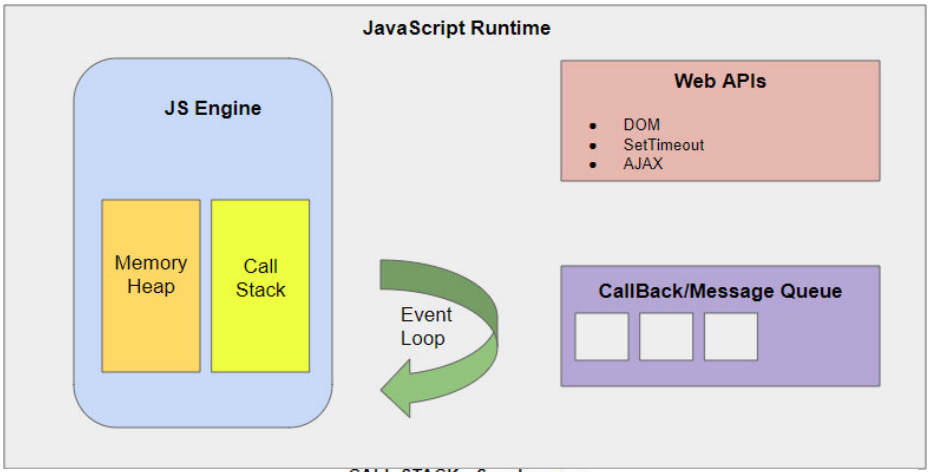
A. The extended class cannot have a method of the same name as the base class

B. The extended class can override the method from the base class

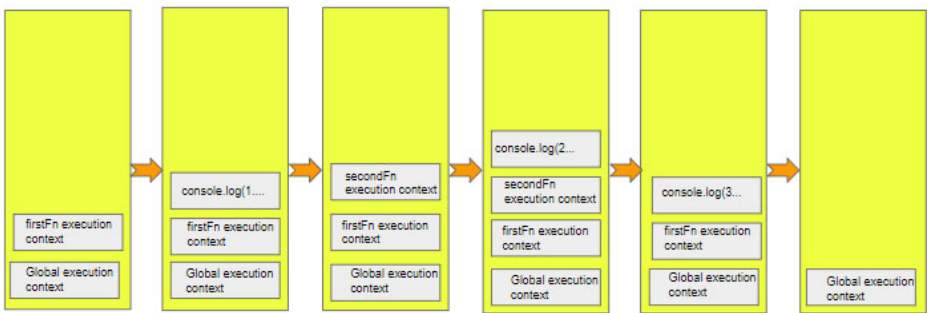
C. The extended class cannot access the constructor of the base class

D. All of the above

Answer: Option B.

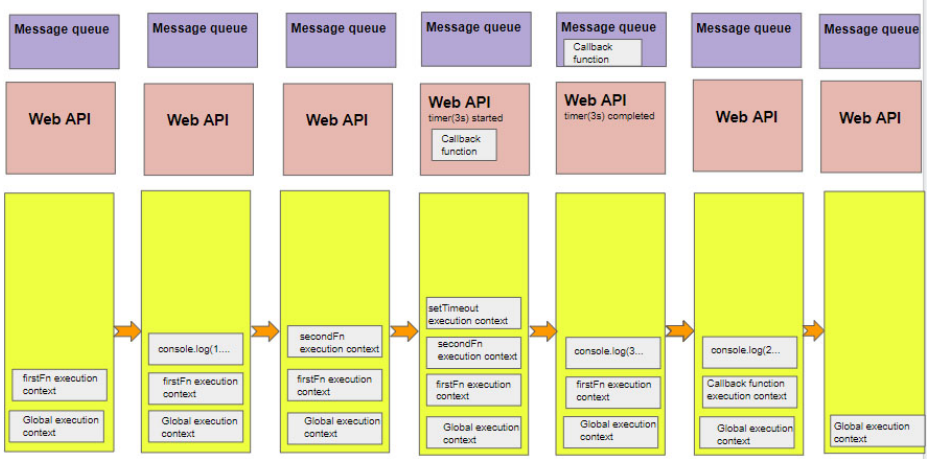


CALL STACK - Synchronous




```
1. Welcome to first function
2. This is the second function
3. This is the end of first function
```

CALL STACK - Asynchronous




```
1. Welcome to first function
3. This is the end of first function
undefined
2. This is the second function
```

```
121 Worcester Rd
Fairhaven
NY 14228
```

```
async function aFunc() {
  return 10;// this becomes an implicit promise
}
aFunc().then((result) => console.log(result)); // 10
async function explicitFunc() {
  return Promise.resolve(10);
}
console.log(explicitFunc()); // shows a promise
explicitFunc().then((result) => console.log(result)); // 10
async function explicitErrFunc() {
    return Promise.reject(new Error('This explicit promise is rejected with error!'));
}
explicitErrFunc().catch(err => console.log(err));
```

```
▸ Promise {<pending>}
10
10
Error: This explicit promise is rejected with error!
    at explicitErrFunc (<anonymous>:11:27)
    at <anonymous>:13:1
▸ Promise {<resolved>: undefined}
```

CHAPTER 7

Asynchronous JS

"Sometimes you are in sync with the times, and sometimes you are in advance, sometimes you are late."

~ Bernardo Bertolucci

In the last few chapters, you learnt about JavaScript and its application with respect to functional programming and object-oriented programming. In this chapter, you will be introduced to another important feature of JavaScript and its runtime engine, the asynchronous JS. Since the inception of programming and computers, there has always been a need and quest for optimization, which always keeps bringing new ideas and new technologies at the forefront. The technology world keeps changing be it something better, faster and more efficient or something totally new, solving an old problem. There was a world of programming languages which promoted multithreaded programming handling multiple problems in parallel to solve them faster, but there was always a need for an idea to avoid the blocking of threads due to a long-running process. If the thread was waiting for some process indefinitely and finally timed out, it only leads to user frustration as he was not able to proceed with anything else. This is the basic problem solved by the idea of Asynchronous Programming.

In this chapter, you will learn what is Asynchronous behavior, how JavaScript Runtime Engine handles synchronous and asynchronous code,

and the different asynchronous constructs of JavaScript, including callbacks, promises and async-await blocks.

Structure

- Asynchronicity in JavaScript
- Callbacks
- Promises
- Async/await

Objective

At the end of this chapter, you will learn all about what is asynchronous programming and how it can be handled in JavaScript applications.

Asynchronicity in JavaScript

Before we begin our journey into the asynchronous world, let's first understand what it means. In any programming language, when the lines of code are executed and completed in the order in which they appear, it's called synchronous flow or behavior.

If you assume the list below to be a set of lines of code which is running synchronously, it will always get executed in the same order. Line 2 will be executed and completed only after Line 1 has completed execution and before Line3 starts execution:

```
Line of code 1;
Line of code 2;
Line of code 3;
```

In asynchronicity, the order in which the lines of code appear does not decide the order in which they complete execution. Line 3 can get completed before Line 1 and 2. Line 1 and Line 2 can get executed at the same time, but Line 2 can finish before Line 1.

Asynchronous programming is a way of programming where lines of code run separately and in parallel to the main application thread and notify the main thread once they get completed with the status being success or failure.

Some operations may take longer to complete, so instead of making the entire execution wait for the operation to complete, they can run asynchronously in the background and send a response once they get completed. This promotes better usage and lesser wait time.

JavaScript runtime

Single-threaded synchronous execution language: This precisely defines the way JavaScript handles execution. JavaScript engine is a program which takes the JavaScript code and executes it. From the beginning of JavaScript when the execution was done by basic interpreters, to the engines available now which are much more performant and effective, JavaScript has come a really long way.

Let's first learn about the different parts of the JavaScript runtime environment to understand the code execution in simpler terms. The JS Runtime includes the following main parts:

1. The JavaScript Engine which takes care of the JS code execution and comprises of:
    - **Memory Heap:** The part of the JS Engine where the memory is allocated for the variables and functions of the code being executed.
    - **Call Stack:** The single-threaded stack where each execution context gets stacked as the code is executed. The event loop continuously keeps checking the call stack, which is a **Last-In-First-Out (LIFO)** queue, to see if there is any function to be executed. It keeps adding any function calls it encounters during executing to the call stack and executes each one in the order of LIFO.
4. The **Web APIs** are in the browser but not part of the JS Engine. They are part of the runtime environment and include asynchronous operations like `setTimeout()` function, DOM manipulation methods, `XMLHttpRequest` for AJAX, Geolocation, LocalStorage.
5. **Callback/Message Queue** is a queue into which the callback, associated with asynchronous calls, are pushed into, after the completion of the associated asynchronous operation. The callback gets picked for execution by the event loop and pushed for execution when the call stack is empty.
6. **Event Loop** is the part which is always running and checking the Call Stack to see if there are any frames to be executed, which it picks for execution. If the call stack is empty, it next checks in the Callback queue to see if any callback is waiting to be executed. if available, it picks the callback from the message queue and pushes into the Call Stack for execution.

The following diagram depicts the various components of the JavaScript Runtime engine:

Figure 7.1: *JavaScript runtime environment*

Now let's look at how the code execution makes use of the JavaScript runtime, specifically the **Call Stack** for both synchronous and asynchronous code:

```
const secondFn = () => {
 console.log('2. This is the second function');
}

const firstFn = () => {
 console.log('1. Welcome to first function');
 secondFn();
 console.log('3. This is the end of first function');
}
firstFn();
```

The preceding code gets added to the Call Stack and executed by the Event Loop as explained as follows:

- The `firstFn` gets called and is added to the call stack as the first execution context.
- The first console log gets executed as the next statement on Call Stack, gets logged and removed from the stack.
- The next statement makes another function call, `secondFn()`, which creates a new execution context for the `secondFn`.
- The first statement of `secondFn`, the console log gets added to the Call Stack, gets executed and removed from the stack.
- As that's the end of `secondFn`, the `secondFn` context gets removed from the stack.
- Finally, the last console statement of the `firstFn` gets executed, and `firstFn` gets completed and removed from the Call Stack.

In this synchronous example, the statements got completed, in the same order as they appeared as shown in the following diagram:

Figure 7.2: *Call Stack execution for a synchronous piece of code*

The console output looks like the following:

Figure 7.3: *Console output for the example synchronous piece of code*

Now we make the same code to behave asynchronously by inducing some delay using `setTimeout` in the `secondFn` as the following code:

```
const secondFn = () => {
setTimeout(() => {
  console.log('2. This is the second function');
 }, 3000);
}
```

```
const firstFn = () => {
 console.log('1. Welcome to first function');
 secondFn();
 console.log('3. This is the end of first function');
}
firstFn();
```

For synchronous code, the code was being executed by the event loop from the call stack. In asynchronous programming, other parts of the JS runtime including the WebAPI and the message queue also get involved.

Let's understand the following steps, with the first few steps being the same as there is no change in `firstFn`:

- The `firstFn` gets called and is added to the Call Stack as the first execution context.
- The first console log gets executed as the next statement on Call Stack, gets logged and removed from the stack.
- The next statement makes another function call, `secondFn()`, which creates a new execution context for the `secondFn`.
- In `secondFn`, there is a `setTimeout` for 3 seconds. The `setTimeout` is a JavaScript method handled by the browser concurrently. When the `setTimeout()` function is called, a new execution context is created and put on top of the stack. The timer is created along with the callback and sits in the Web API in a separate thread which keeps running there for 3 secs asynchronously, without blocking the main code flow.
- The `setTimeout()` function returns and is cleared from the Call Stack. And so is the `secondFn()` function.
- The last console statement of the `firstFn` gets executed, and `firstFn` gets completed and removed from the Call Stack.
- Now after the 3 secs have passed, the timer disappears from Web API, and the associated callback function is moved to the Message Queue. It waits there for the execution stack to be empty to be picked up by the Event Loop.
- The Event Loop will constantly monitor the message queue and the Execution Stack. When the execution stack is empty, it pushes the callback from the Message Queue to the execution stack. So a callback execution context is created and the console log executed, completed and moved out of the stack:

Figure 7.4: *JS Runtime for an asynchronous piece of code*

Now if you execute the preceding code in your browser console, the output will be as below, where the last log appears after some delay due to the

asynchronous `setTimeout`:

Figure 7.5: *Console output for the example asynchronous piece of code*

Now that you understand the asynchronous behaviour, the need for it, and how it is handled by the JS Engine, in the next sections, you will learn about the different asynchronous constructs in JavaScript.

Callbacks

Asynchronous programming ensures that the main thread of the program is not blocked waiting for a long-running process to complete. The most basic solution to handle asynchronous behaviour is **Asynchronous Callbacks**. The `callback` function is a parameter passed to the asynchronous function. The `callback` function contains the logic which is to be executed after the asynchronous function gets completed:

```
const getAsyncRespone = callback => {
 setTimeout(() => {
   callback ('This is the response from my asynchronous
   function');
 }, 2000);
}
```

In the preceding example, the `getAsyncResponse` function is the asynchronous function which waits for a duration of 2 seconds and then sends the response. It takes a `callback` function as a parameter which is invoked with the response:

```
getAsyncRespone (response => {
 console.log(response);
})
```

When calling the function as previously, we need to pass a function as a parameter, which in this case is defined as an anonymous function using the arrow syntax and which has the logic to log the response received.

The same can be defined using a named function as follows:

```
function logging(response) {
 console.log(response);
}
getAsyncRespone(logging);
```

The `callback` function named logging is passed as a parameter, without providing the () as it is not being invoked here, but a reference is passed to the main asynchronous function where it will get invoked.

The approach (named or not) does not matter, and the behaviour is the same. The main thing to understand is passing a `callback` function as a

parameter which gets added to the message queue once the asynchronous function gets completed and gets picked by the event loop once the call stack is empty.

The syntax of callbacks involves some nesting, and with every new callback added, an additional level of nesting gets included, which makes the code complex and confusing.

The following code is to print three lines of an imaginary address one after the other, with a delay of 2 seconds for each line:

```
const printAddr = (string, callback) => {
 setTimeout(() => {
console.log(string);
   callback ();
 }, 2000);
}

const getAddress = () => {
 printAddr("121 Worcester Rd", () => {
  printAddr("Fairhaven", () => {
   printAddr("NY 14228", () => {})
  })
 })
}
getAddress();
```

The preceding code is a simple function showing some imaginary address, called at the different callback levels to see how nested the code becomes with a callback. You can imagine how a more complex code with multiple callback levels would look like. The console output of the code block with callbacks is as follows:

Figure 7.6: *Console output for the callback example*

Think of another example of cooking a pizza where each of the actions is independent methods implemented separately and with some inputs and outputs:

```
const makeAPizza = () => {
 makeTheBase((base) => {
  applySauce((sauce) => {
   addToppings((toppings) => {
  bakeIt((pizza) =>
{
return pizza;
})
})
```

```
    })
  })
}
myPizza= makeAPizza();
```

Here each asynchronous step (as it takes some time) should be completed before you proceed with the next one.

Promises

Promises are one of the most important enhancements to JavaScript as part of ES6. We will read about the other ES6 changes in the next chapter, *Chapter 8*, What's New in ES2019 JavaScript.

Promises are constructs which handle asynchronous behaviour in a much cleaner way when compared to callbacks.

The ECMA Committee defines a promise as:

"A Promise is an object that is used as a placeholder for the eventual results of a deferred (and possibly asynchronous) computation."

When a task is supposed to take some time and will send response eventually, it can be tracked with a promise. Promise is like a real promise you make, which will get the response back whenever it is available.

A promise is created as shown below using the Promise constructor.

```
const myNewPromise = new Promise((resolve, reject) => {
 if (Math.random() * 10 <= 6) {
      resolve('Hello, My promise was a success!');
 }
 reject(new Error('My promise failed'));
});
```

A promise is related to an activity which is taking time, it also undergoes some changes in state as it is waiting for a response and finally, when it receives the response.

The different states of a promise include:

- **Pending:** This is the state while the promise is waiting for the event to complete, so the promise is said to be in the pending state.
- **Settled/Resolved:** Once the asynchronous event has completed, the promise has received the result and is said to be settled or resolved. Depending on the type of result it receives, the promise is said to be:
    - **Fulfilled**: When the event completed successfully, the promise is successful, and the response is available, it is said to be fulfilled.
- **Rejected:** If there was an error during the process, then the promise

is said to be rejected.

As you saw in the `myNewPromise` example, the promise constructor takes a callback as an argument. The `callback` function takes two parameter functions which are used when the promise finally completes:

- `resolve(value)`: This function is used to send the success response for the promise:
  - This will set the Promise Status to `fulfilled`.
  - The promised value will be set to the value sent as a part of this function.
- `reject(error)`: This function is used to send the error response for the promise:
  - This will set the Promise Status `to rejected`.
  - The promised value will be set to the error passes as an argument to this function.

Now that you understand how to create a promise which will get settled and get converted to either fulfilled or rejected, the next step is to consume the promise.

To consume a promise, we would require two sets of handlers, for fulfilled state and for the rejected state. The promise is consumed using the `then()` method.

There are few possible ways this can be done syntactically, and we will see each of these here using the `myNewPromise` example.

1. **then() accepts two callbacks:**
   - The first `callback` function is executed when the promise is resolved or settled.
   - The second `callback` function is invoked when the promise is rejected.

```
myNewPromise.then((success) => {
  console.log(success);
}, (error) => {
  console.log(error);
});
```

4. **The two parameters are optional and can be selectively handled:**
   - Only handle success:

```
myNewPromise.then((success) => {
  console.log(success);
});
```

- Only handle error

```
myNewPromise.then(null, (error) => {
  console.log(error);
});
```

7. The chaining of catch to handle the error:

```
myNewPromise.then((success) => {
console.log(success);
 })
.catch(error => console.log(error));
```

By chaining the then and catch methods, any promise can be handled.

Also, if there is a need to call multiple promises, it can be chained using the then method any number of times and any number of levels.

Let's implement the same dummy address example we used for callbacks, using promises.

First create the promise to log the address sent after a delay of 2 seconds using `setTimeout`:

```
const printAddr = (string) => {
promise= new Promise( (resolve) => {
 setTimeout(() =>{
console.log(string);
resolve(string);
}, 2000);
}
);
return promise;
}
const getAddress = () =>
 {
  printAddr("121 Worcester Rd").then(()=>{
  return printAddr("Fairhaven");
}).then(()=>{
  return printAddr("NY 14228");
}).catch((err) =>{
console.log(err);
});
}
getAddress();
```

This example shows how promises can be chained so that the response from one promise can be passed on to the next using then and catch blocks.

This is a better structured and segregated code than compared to callbacks,

and you can notice the difference of structure and clarity as to the size of the code increases.

But the best is yet to come.

Async Await

Promises did solve the problem with callbacks, but the code still looked big and complex. In ES8 JavaScript, a totally new asynchronous construct was introduced, which made it easier to work with promises—Async Await.

Both the keywords, async, await, are placed separately but work together to handle asynchronous behaviour.

Let's understand how we can use them and how the code looks this time.

Async

The async keyword, when placed before a function, indicates that the function will behave asynchronously and will always return a promise implicitly or explicitly.

Asynchronous function using the async keyword:

```
async function aFunc() {
 return 10;// this becomes an implicit promise
}
// Async function will always be handled with then/
catch as it returns a promise.

aFunc().then((result) => console.log(result)); // 10
```

It can return a promise explicitly as shown:

```
async function explicitFunc() {
 return Promise.resolve(10);
}
console.log(explicitFunc()); // shows a promise
explicitFunc().then((result) => console.log(result));
// 10
```

It can also resolve into an error as shown:

```
async function explicitErrFunc() {
 return Promise.reject(new Error('This explicit promise
 is rejected with error!'));
}
explicitErrFunc().catch(err => console.log(err));
```

The console output of the async function code:

Figure 7.7: *Console output for the async example*

Irrespective of what is within an `async` function, it will return a promise, either implicitly or explicitly.

Await

The await operator is used to wait for a promise inside an `async` function. It is used before any asynchronous statement within the `async` function, and it stops the flow at that point to wait for its completion. It brings out synchronous behaviour within asynchronous behaviour. The response of an await statement will always be a promise.

Now, we will define an asynchronous function which returns an explicit promise as follows:

```
const printAddr = (string) => {
promise= new Promise( (resolve) => {
 setTimeout(() =>{
console.log(string);
resolve(string);
}, 2000);
}
);
return promise;
}
```

The `promise` function is the same as the last example, and the difference would be how we consume it.

The function is marked as `async` as it has asynchronous statements.

It is calling the asynchronous function `printAddr` three times as you can see in the three await statements.

Now the following code shows how async and await work together:

```
async function getAddress(){
var first = await printAddr("121 Worcester Rd");
var second= await printAddr("Fairhaven");
var third= await printAddr("NY 14228");
return third;
}
getAddress().then((response)=>{
console.log(response);//The third will be printed twice
due to this
});
```

The `await` will result in the first promise getting completed, only then the execution will go to the next line to fetch the second promise and

similarly the third after the second is fulfilled. This is a sequential behaviour within the `async` function imposed by the use of await. If we do not want the three calls to be made in sequence as there is no internal dependency, instead make the calls in parallel. We can achieve parallel processing of the asynchronous calls by making use of `Promise.all` as shown as follows:

```
async function getAddress(){
var [first, second, third] =
 await Promise.all([printAddr("121 Worcester Rd"),
 printAddr("Fairhaven"), printAddr("NY 14228")]);
return third;
}
getAddress().then((response)=>{
console.log(response);//The third will be printed twice
due to this
});
```

This code definitely looks cleaner and better than multiple callback loops and multiple then catch chains. It also provides us with the capability to execute the statements synchronously or asynchronously within the `async` function by making use of await and `Promise.all`.

Let's look at different scenarios of making use of `async-await`.

Scenario 1 – Synchronous behavior within asynchronous

Function `firstFn` returns a `Promise`, and then function `secondFn` needs the value returned by function `firstFn` and function `thirdFn` needs the resolved value of both functions `firstFn` and `secondFn`.

The following code shows an example for the scenario:

```
async function asyncTaskSynchronously () {
 const value1 = await first()
 const value2 = await second(value1)
 return third(value1, value2)
}
```

This behaves in a synchronous way as each step is dependent on the result of the previous step and cannot proceed.

You can think of writing this using other possible ways, and undoubtedly nothing is as simple!

Scenario 2 – Parallel execution within asynchronous

Function `firstFn`, function `secondFn` and function `thirdFn` return promises, then the function `fourthFn` needs all the three values returned

by function `firstFn`, `secondFn` and `thirdFn` for further computations. There is no internal dependency between `firstFn`, `secondFn` and `thirdFn` functions, and they should run in parallel to avoid any unnecessary wait time.

The following example code shows the use of `async-await` to achieve parallel execution within asynchronous:

```
  async function AsyncTaskParallely () {
const [ value1, value2, value3 ] = await Promise.all([
firstFn(), secondFn(), thirdFn() ])
 fourth(value1,value2,value3)
}
```

Whenever you want to run asynchronous operations in parallel without causing any blocking and intermediate wait time, you should always wrap them into `Promise.all()`. This combines the power of `async-await` with promises.

Scenario 3 – Async with classes

Class `myClass` has a function `firstFn` which is asynchronous and returns a promise. The next piece of code shows an example of `async-await` with classes.

```
class myClass {
async firstFn() {
return await Promise.resolve('first resolved');
}
}

new myClass()
.firstFn()
.then(alert); // 'first resolved'
```

Await can be used along with any asynchronous function call. It should be used within a function which is defined using `async`. The `await` keyword cannot be used directly at top-level without being included inside an `async` function.

Scenario 4 – Error handling for async-await

Error handling in `async-await` block can be done as any chain of promises using the `try…catch block` as shown in following example:

```
const asyncFnCall = async () => {
try {
const data = JSON.parse(await getJSON())
const another = await getData(data)
console.log(data, another)
```

```
} catch (err) {
console.log(err)
}
}
```

Apart from using the `try…catch`, it can also be handled using an overall catch as follows:

```
asyncFnCall ().catch(alert);
```

If you forget to handle the error explicitly by including The `.catch` here also, you will get an unhandled promise error in the console.

Scenario 5 – Working with arrays asynchronously

Working with an array of items needs to be taken care of specifically, whether or not it should behave synchronously or asynchronously while handling each item of the array. Let's look at an example to perform a set of asynchronous operations, like reading a set of files and logging the data.

- To perform the asynchronous operations in sequence using `async/await` as shown as follows:

```
async function printFiles () {
 const files = await getFilePaths();

for (let file of files) {
 const contents = await fs.readFile(file, 'utf8');
 console.log(contents);
 }
}
```

- To perform the set of asynchronous operations like reading the files, in parallel can be done as shown as follows:

```
async function printFiles () {
 const files = await getFilePaths();

await Promise.all(files.map(async (file) => {
 const contents = await fs.readFile(file, 'utf8')
 console.log(contents)
 }));
}
```

You saw different scenarios of how you can use async-await to handle asynchronous behaviour by writing much less and simpler code. You should be careful in understanding how you want to use it by making use of parallel execution wherever possible and reducing any unnecessary wait times.

Conclusion

In this chapter, you were introduced to the concept of asynchronicity and how asynchronous behaviour is handled in JavaScript using `callbacks`, `promises` and `async-await` constructs. Whenever a task takes an uncertain time, it should be handled asynchronously instead of the thread waiting for the task to complete and blocking the complete flow.

Having introduced to the different aspects of programming in JavaScript functional, object-oriented and using asynchronous constructs, you can structure your code in the most efficient way.

In the next chapter, you will be learning about some of the recent and important ECMAScript changes which have added extra capabilities to JavaScript, specifically the ECMA versions, ES6 and ES9. These additions have made JavaScript more powerful.

Questions

1. The await can be used without async to indicate asynchronous function call.

   A. TRUE

   B. FALSE

   Answer B.

   The await cannot be used without async to indicate asynchronous function call. It can be used in a function which is marked as async only.

2. The promise changes state as it waits and later when it is completed. What are the different statuses?

   A. pending, fulfilled, rejected

   B. pending, succeeded, errored

   C. waiting, fulfilled, rejected

   D. Promise does not have any status.

   Answer A.

   Promises are in the pending state while it is waiting for the event to complete, fulfilled state when it completes successfully, rejected when it completes with an error.

3. ____ are part of the runtime environment and include asynchronous operations like setTimeout() function, DOM manipulation methods.

   A. Call stack

   B. Web APIs

C. Message queue

D. Event loop

Answer B.

The Web APIs are in the browser but not part of the JS Engine. They are part of the runtime environment and include asynchronous operations like setTimeout() function, DOM manipulation methods, XMLHttpRequest for AJAX, Geolocation, LocalStorage.

4. Functions fnA, fnB and fnC return Promises and have no internal dependency, How should await be used?

   A. const valueA = await fnA(); const valueB = await fnB(); const valueC = await fnC();

   B. const valueA, valueB, valueC = await fnA(), fnB(), fnC();

   C. const [valueA, valueB, valueC] = await Promise.all([fnA(), fnB(), fnC()]);

   D. This is not a valid use case to use async await.

   Answer C.

   `Promise.all`, in combination with await, can be used to parallelly wait on multiple promises which are not dependent on each other.

5. The advantages of using async-await include

   A. Clarity

   B. Readability

   C. Reduced complexity

   D. All of the above

   Answer D.

Stage 0	Stage 1	Stage 2	Stage 3	Stage 4
Strawperson	Proposal	Draft	Candidate	Finished
Allow input into the specification	Make the case for the addition and describes the solution after identifying any potential challenges.	Describe the syntax and semantics	Enhancements based on feedback	Ready for inclusion

```
console.log(greetFn('Hello '));
```

```
Hello JavaScript
```

```
let num=[10,20,30];
square= num.map((n,index)=> n*n);
console.log(square);
```

```
▶ (3) [100, 400, 900]
```

```
let num1=[10,20,30];
square= num1.reduce((sum,n)=> sum + n*n);
console.log(square);
```

```
1310
```

```
let num2=[10,20,30, 40,50,60];
filtered= num2.filter((num)=> num>=30);
console.log(filtered);
```

```
▶ (4) [30, 40, 50, 60]
> const myString ='Welcome to the world of JavaScript';
< undefined
> myString.startsWith('Welcome');
< true
> myString.startsWith('Hello');
< false
> myString.endsWith('JavaScript');
< true
> myString.endsWith('Java');
< false
> myString.repeat(2);
< "Welcome to the world of JavaScriptWelcome to the world of JavaScript"
>  message = `Welcome to the world of  ${name}! ${name} is exciting!`;
< "Welcome to the world of  JS! JS is exciting!"
>  message = `Welcome to the world of  ${name}!
            ${name} is exciting!
            ${name} is fun`;
< "Welcome to the world of  JS!
            JS is exciting!
            JS is fun"
```

CHAPTER 8

What's New in ES2019 JavaScript

"If we want users to like our software, we should design it to behave like a likeable person: respectful, generous and helpful."

— Alan Cooper, Software Designer and Programmer

Having started from the basics of JavaScript and having gone through the functional, object-oriented and asynchronous aspect of JavaScript, we will explore the latest features of JavaScript and understand how we can use them in our day-to-day programming in this chapter. We will go through the ES6 changes as it was a major release which has been well adopted and we will look at the latest version changes as part of ES10(ES2019).

Structure

- ECMAScript standardization
- Essential ES6 changes

- ES2019 JavaScript – Major new features
- ES2019 JavaScript – Minor new features

Objective

At the end of this chapter, you will become aware of the latest enhancements in JavaScript through ECMAScript.

ECMA Script Standardization

Every year, a new version of ECMAScript is released with proposals which have been reviewed, accepted, and reached stage four by the TC39 committee. Some of them would be included in the specification of that year, and the others are postponed to the next year. We read about this in *Chapter 1, History of JS* and how it has revolutionized web development which elaborated the process and the different versions and features which have come out over the years since its inception in 1996-97.

The following diagram shows the stages of the TC39 committee:

Figure 8.1: *TC39 Standardization stages*

In this chapter, we will look at some of the important features introduced in recent years which have changed the initial face of JavaScript.

Essential ES6 changes

Before going through the latest ES10 changes of ECMAScript, we will first go through the important ES6(ES2015) changes which brought some useful constructs into the JavaScript language. This version brought in some major changes; much of it was syntactic sugar but which was much awaited and have been well accepted by the JavaScript user community. These are extensively used in ReactJS, which you will be introduced to in later chapters, so it's essential that you gain a good understanding of these JS changes.

Handling variables from var to const/let

You were introduced to `let` and `const` in the *Chapter 4, Introduction to JavaScript Variable Declarations,* as the block scoped variable declarations. These are a relatively recent change in JS which came in ES6, but it is a better way of declaring variables instead of going the var route. Hence, instead of waiting to introduce `let` and `const` at this point, it was introduced in the very beginning so that you learn a better way to start with.

Just to recap, `let` and `const` variables are valid within the block of

curly braces in which they are declared. The `let` variable can be reassigned and changed later, but the const variable cannot be reassigned and behaves like a single value constant.

The power of arrow functions

A much awaited feature, which ends all problems with the `this` keyword and closures, the arrow function. It makes the code compact, easy to read, and understand.

The variations of the syntax of the arrow function are as follows:

```
//with no parameters and single statement which is the
automatically returned value with no return keyword
constfunctionName= () =>console.log('Learning JS');
functionName();// Learning JS

//With n1, n2, n3…parameters and single return
statement
constsumFn= (n1,n2,n3) =>n1+n2+n3;
sumFn(1,2,3);// 6

//With n1, n2, n3…parameters and multiple lines of code
within {} and explicit return statement
constlongSumFn= (n1,n2,n3) => {
const sum = n1+n2+n3;
return sum;
}
longSumFn(1,2,3);// 6
```

The traditional function is written as follows:

```
function greetFn(message){
const name = 'JavaScript';
const greeting = message + name;// this forms the
greeting
console.log(greeting);
}
```

It can be written as follows:

```
constgreetFn = message =>{
const name = 'JavaScript';
const greeting = message + name;// this forms the
greeting
return greeting;
}
```

Or in a single line(with whatever needs to be returned):

```
constgreetFn = message => message + 'JavaScript';
```

The function can be called as before by passing the required parameter as shown:

Figure 8.2: *Console output for the arrow function*

We saw the use of `arrow` function when working with arrays in *Chapter 4,JavaScript Programming – Making the Application Interactive*, using functions like map, filter, and reduce on arrays.

A quick recap of the same can be seen using the short syntax (single line) of the arrow function as follows:

Figure 8.3: *Example showing map, reduce and filter with the arrow function*

The most important advantage of using arrow functions is that unlike regular functions, they don't get their own this keyword. Instead, they automatically use this keyword value of the parent function they are written in.

Handling strings

ES6 brings along some new string methods which make some string manipulation easier as follows:

- `booleanstartsWith(string)`
- `booleanendsWith(string)`
- `boolean includes(string)`
- `string repeat(count)`

Figure 8.4: *Example of ES6 new string methods*

Template literals

Talking about strings, another new addition in handling strings is template literals. In traditional JS, if we had to include variables in a string, we would use concatenation with + like as follows:

```
const name = 'JS';
const message = 'Welcome to the world of '+ name +'!' +
name + 'is exciting!';
```

You can imagine the pain if the size and number of variables increases. The template literal provides a cleaner way of doing this concatenation with variables.

The entire string is within back ticks (‘) and the variable is embedded within as interpolation with this syntax `${variable}`.

The code looks like the following:

```
message = `Welcome to the world of ${name}! ${name} is
exciting!`;
```

This can also be used if the string extends to multiple lines without having to break it into multiple strings; simply use backticks instead:

```
 message = `Welcome to the world of ${name}!
    ${name} is exciting!
${name} is fun`;
```

The console looks like the following:

Figure 8.5: *Example of using template literals*

Handling parameters

ES6 provides the capability to provide a default parameter value in your functions. The default value for the parameter can be given using an equal sign (=). When the function is called by providing a value for the parameter, then the provided value is used, but if no value is passed to the parameter while calling the function, the default value is used. In the following example, the default parameter value of `num2` is `10`:

```
function MyFunction(num1, num2=10) {
 return num1+num2;
}
MyFunction(1, 2); //3
MyFunction(10); // 20
MyFunction(); //NaN
```

In the first usage, the default value is not used as both the parameters are provided.

In the second case, the second parameter is not provided; hence, the default value gets picked up.

In the third case, none of the parameters are provided, but since there is no default defined for the first parameter, it is undefined which results in the sum being NaN.

From arguments to rest parameters

If the number of parameters is not fixed, ES6 provides a good solution to handle this scenario in the form of rest parameters denoted by '…'. By prefixing a … to a parameter, it will take all the remaining parameters into it in the form of an array.

Let's look at an example to understand this further:

```
function RestFn(num1, …nums) {
console.log(num1, nums);
```

```
 return num1 *nums.length;
}
RestFn(10, 20, "helloJS", true, 50);
```

The output is shown in the following screenshot:

Figure 8.6: *Rest example output*

Here `num1` takes the first argument, whereas all the remaining arguments comprise the array in spite of being of different data types.

Symbols

The new primitive data type called **symbol** was included as part of ES6. Symbols always generate a unique value and hence can be used as follows:

- To create unique identifiers for object properties
- To create unique constants

```
constSYM_CONSTANT = Symbol();
let objSym = {};
objSym[SYM_CONSTANT] = 10;//10
```

Spread operator

The spread operator (…) is very useful and helps to expand elements of an array and assign the direct elements of the array to functions arguments, other arrays, or wherever we need to consume the array elements:

```
let arr1= [1, 2, 3, 4 ];
```

If you want to take the elements of the array and assign it to another variable, you can use the spread operator as follows:

```
let arr2= […arr1, 5,6]; // [1, 2, 3,4,5,6]
let arr3= […arr1, 8, …arr2];//[1, 2, 3, 4, 8, 1, 2, 3,
4, 5, 6]
```

Destructuring

Destructuring is another useful way of extracting data from arrays and objects. The destructuring syntax does the assignment on the left-hand side to define what values to be unpacked from the source variable:

```
let arr1 = ['A', 12];
const [name, age] = arr1;// arr1 is destructured into
name and age.
console.log(name);//A
console.log(age);//12
```

```
// Default values while destructuring, in case value
unpacked from the array is undefined.
// Also some values can be ignored by giving
placeholder commas
const [name = 'A',, age = 12] = ['A', 'R', 12];
console.log(name);
console.log(age);
//Values can be swapped
let a = 100, b = 200;
[b, a] = [a, b];
console.log(a, b);//200, 100
```

Destructing can also be applied to objects:

```
constobj= {
firstName: 'Virat',
lastName: 'Kohli'
};
// Variable names must match object keys at the time of
destructuring
const {firstName, lastName} = obj;

console.log(firstName);// Virat
console.log(lastName);// Kohli
```

Classes

Classes are perhaps the most awaited syntactic sugar, a nicely presented syntax, to provide the object-oriented behavior in JS. In traditional JavaScript, classes were created using functions and then came the class keyword as part of ES6 changes as we learned in *Chapter 6, Object Oriented JavaScript*.

To recap what we already learned in *Chapter 6, Object Oriented JavaScript*, with the arrival of classes in ES6, you can define classes with properties and methods as follows:

```
class System{
  constructor(name) {
this.name = name;
this.type ='construction';
  }
getName() {
   return this.name;
  }
getType(){
   return this.type;
```

```
}
}

// An instance can be created using the new keyword
let constructSys= new System('robot');
```

When the instance is created with the new keyword, the constructor is invoked.

A class can be further extended to inherit the properties and methods of the base class and enhance it with additional properties and methods:

```
class Alarm extends System{
  constructor(name) {
   super(name);
this.type = 'alarm'
  }
}

let AlarmSys = new Alarm('water-level');
AlarmSys.getType();//"alarm"
AlarmSys.getName();//"water-level"
```

The constructor of the base class can be invoked using super() to ensure the base behavior is retained.

In this section, we went through all the major ES6/ES2015 changes. Next, we will get into the latest ES10/ES2019 changes.

ES2019 JavaScript – Major new features

In this section, we will learn about the features of ES2019, the latest ECMAScript version, and how you can start using them in your regular programming.

Array.prototype.{flat,flatMap}

This comprises two new methods to work with arrays, especially nested arrays.

Array.flat()

The `flat()` method is an array method which creates a new array with all sub-array elements concatenated into it at the same level, recursively up to the specified depth. This can be applied to nested arrays to any level to flatten it up into a simpler array:

```
var arr1=[1,2,3,4,[5,6,7,[8,9,10,[11,12,13,[14]]]]];
arr1.flat() or arr1.flat(1) will perform 1 level of
```

```
flattening and the output will be
[1,2,3,4,5,6,7,[8,9,10,[11,12,13,[14]]]];
arr1.flat().flat() or arr1.flat(2) will perform 2
levels of flattening and the output will be
[1,2,3,4,5,6,7,8,9,10,[11,12,13,[14]]];
```

This can be continued for any number of levels. If the number of levels is unknown, but we want to perform complete flattening, we can use the following:

```
arr1.flat(Infinity) //
[1,2,3,4,5,6,7,8,9,10,11,12,13,14]
```

Array.flatMap()

The `flatMap()`, another array method, first maps each element using a mapping function, and then flattens the result into a new array. It is similar to applying a map function followed by a flat function of depth 1. If map returns a nested array, `flatMap` consolidates and creates a single level array:

```
let arr2=[1,2,3,4,5,6];
arr2.map(x=>[x,x*x]); // returns a list of nested
arrays with number and its square
[[1,1],[2,4],[3,9],[4,16],[5,25],[6,36]]
arr2.flatMap(x=>[x,x*x]);
[1,1,2,4,3,9,4,16,5,25,6,36]//the list will be
flattened at one level.
```

The result map arrays will be combined and flattened at one level.

Object.fromEntries

A new object method will take a list of key-value pairs of data and convert it into a single object. It is the reverse of what `Object.entries` does. Let's see with the following example:

```
let objectClass = { girls: 20, boys: 15};
//applying Object.entries on the above Object will give
an Array of key, value pairs
let entriesClass = Object.entries(objectClass );
entriesClass;// [["girls",20],["boys",15]]
```

Now, we can reverse and convert the above array to an object using `Object.fromEntries` as shown in the following code:

```
finalObj = Object.fromEntries(entriesClass);
finalObj; // gives the object { girls: 20, boys: 15}
```

```
This method can be applied to any similar Array or map
to convert into an Object.
arrayName=['First','Second','Third']
age=[11,12,13];
mapped =arrayName.map((each,index) => [each,
age[index]]);
console.log(mapped);//[["First", 11],["Second", 12],
["Third", 13])
final= Object.fromEntries(mapped);// {First: 11,
Second: 12, Third: 13}
```

Finally, we will get the object with key-value pairs.

ES2019 JavaScript – Minor new features

The other new features which are part of this latest release include the following listed minor features.

String.prototype.{trimStart,trimEnd}

This includes two string methods which are used to remove any additional white spaces at the start or end of the string.

String.trimStart()

The `trimStart()` method trims or removes the white spaces at the start of a string. This is alias of `trimLeft()` and does the same thing as shown in the following code:

```
let message= " I have spaces on both sides ";
message.trimStart(); // " I have spaces on both sides
";
message.trimLeft(); // " I have spaces on both sides ";
```

String.trimEnd()

The `trimEnd()` method trims or removes the white spaces at the end of a string. This is alias of `trimRight()` and does the same thing as shown in the following code:

```
let message= " I too have spaces on both sides ";
message.trimEnd(); // " I too have spaces on both
sides";
message.trimRight(); // " I too have spaces on both
sides ";
```

Symbol.prototype.description

Symbol is a primitive data type, introduced as part of ES6, which creates a unique value every time you use `Symbol()` to create one.

It can be used as key identifiers for object values. A read-only optional description property can be provided for the `Symbol` at the time of creation. It can be used to retrieve the property for debugging and logging.

`Symbol` can be created with and without the description property as shown in the following code:

```
let mySymbol1= Symbol();//without description
let mySymbol2= Symbol("My symbol with description
property");
mySymbol2.description;// "My symbol with description
property"
mySymbol1.description;//undefined as no description was
provided
```

Optional catch binding

There is a change in the try-catch syntax in the ES10 JS version.

Earlier, in order to have a `catch` block, it was mandatory to provide a parameter in the `catch` block to handle the error as shown in the following code:

```
try{
 throw new Error("My Custom Error");
}catch(error){
console.log("The error caught", error);// The error
caught Error: My Custom Error
}
```

With ES10, the parameter for the `catch` block is now optional and it is shown as follows:

```
try{
 throw new Error("My Custom Error");
}catch{
console.log("The error caught");// The error caught
}
```

New Function.toString()

The `toString()` method when applied to a function returns a string which represents the source code of the function. Earlier, this would remove any white spaces, new lines and comments in the function and return it in a not very reader friendly format. With the latest changes, now all the white spaces, new lines, and comments are retained in the original

source code and it returns in a proper format as shown in the following code:

```
function greetFn(message){
const name = 'JavaScript';
const greeting = message + name;// this forms the
greeting
console.log(greeting);
}
```

Figure 8.7: *Console output of Function.toString()*

JSON ⊂ ECMAScript (JSON Superset)

According to ECMAScript specification, an ECMAScript JSON object is a superset of JSON.

As per the TC39 ECMAScript specification, *"After parsing, JSON objects are realized as ECMAScript objects. JSON arrays are realized as ECMAScript Array instances. JSON strings, numbers, Booleans, and null are realized as ECMAScript strings, numbers, Booleans, and null."*

Previously, the line separator (U+2028) and the paragraph separator (U+2029) symbols were not allowed as valid ECMAScript strings as they were considered valid JSON strings. This lead to inconsistency and errors on parsing JSON to ECMAScript. As per the latest change, this inconsistency is resolved and the line separator (\u2028) and paragraph separator (\u2029) symbols are included as valid string literals in ECMAScript as follows:

Figure 8.8: *Console output of line and paragraph separator*

Well-formed JSON.stringify()

Earlier, when processing, UTF-8 code points (U+D800 to U+DFFF) in JSON using **`JSON.stringify()`**, it would return a malformed unicodecharacter resulting in unknown issues difficult to debug.

This has been fixed in the ES10 version as these are represented using `JSON.stringify` and can be converted back to the original form using `JSON.parse()`.

Stable Array.prototype.sort()

The earlier implementation of `Array.sort()` was not stable as it used quick sort. With the latest changes, the sorting algorithm is more stable now.

"A stable sorting algorithm is when two objects with equal keys appear in the same order in the sorted output as they appear in the unsorted input."

Conclusion

At the end of this chapter, we learned all the important concepts of JavaScript, including the latest ES2019 changes. After gaining comfort on HTML, CSS and JavaScript programming, our next step would be to build a full-fledged application putting our learning to use. In the next chapter, we will be doing just that, building a complete application using HTML, CSS, and JavaScript.

Questions

1. A new Object method which will take a list of key-value pairs of data and convert it into a single object:

    A. Object.entries

    B. Object.fromEntries

    C. Object.new

    D. None of the above

    Answer B

    Object.fromEntries is a new Object method, part of ES2019/ES10 which will take a list of key-value pairs of data and convert it into a single object. It is the reverse of what Object.entries does.

2. By prefixing a … to a parameter, it will take all the remaining parameters into it in the form of an array. What is this concept called?

    A. rest

    B. spread

    C. destructuring

    D. default

    Answer A

3. This block scoped keyword declaration is used whenever the containing value can mutate and undergo change.

    A. constant

    B. const

    C. var

    D. let

    Answer D.

4. Which of the following statements are true about arrow functions?

    A. Arrow functions make the code compact and clear.

B. Arrow functions have two syntax representations for single and multi-line functions.

C. Arrow functions use this keyword value of the parent function they are written in.

D. All the above

Answer D.

All the statements are true with respect to the arrow functions.

5. This is a primitive data type which generates a unique value whenever created.

A. Symbol

B. Enum

C. Array

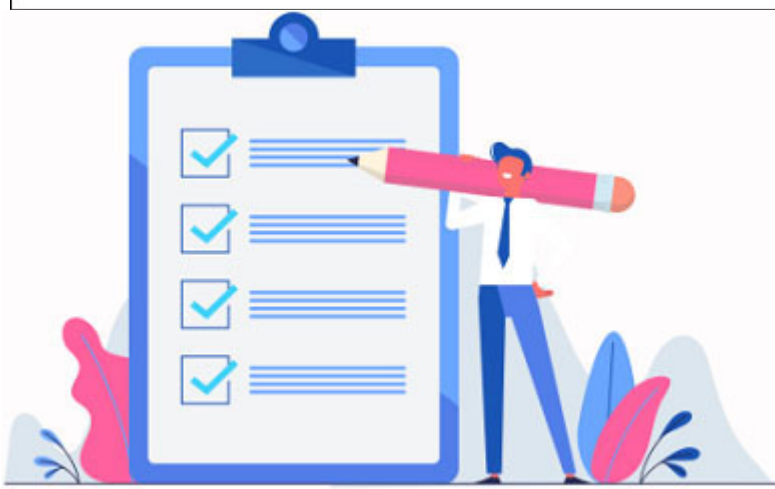
D. Float

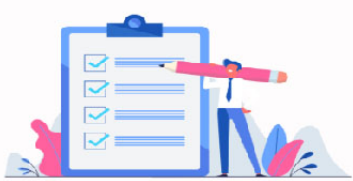
Answer A.

TODO

+ Add Task

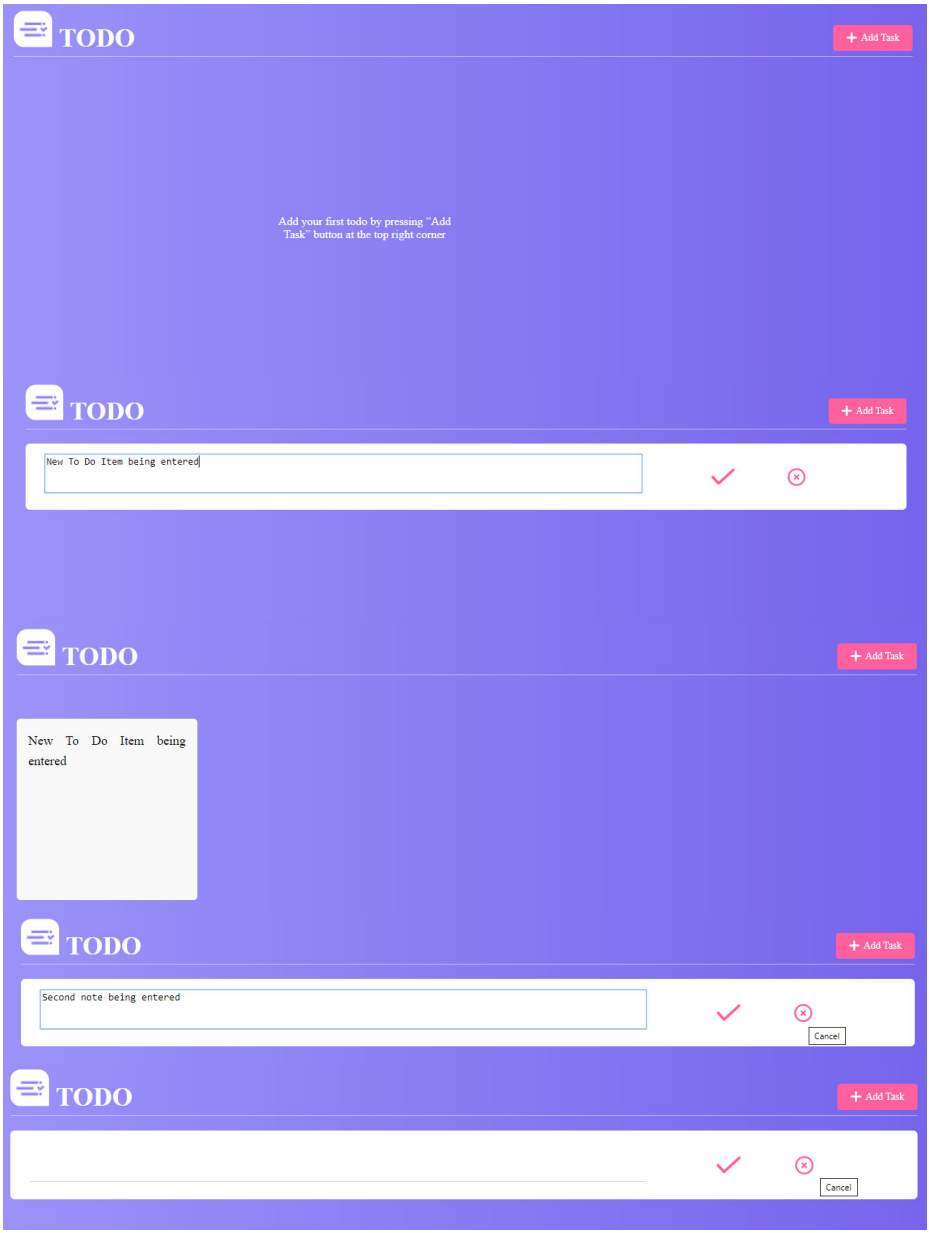
Your text goes here

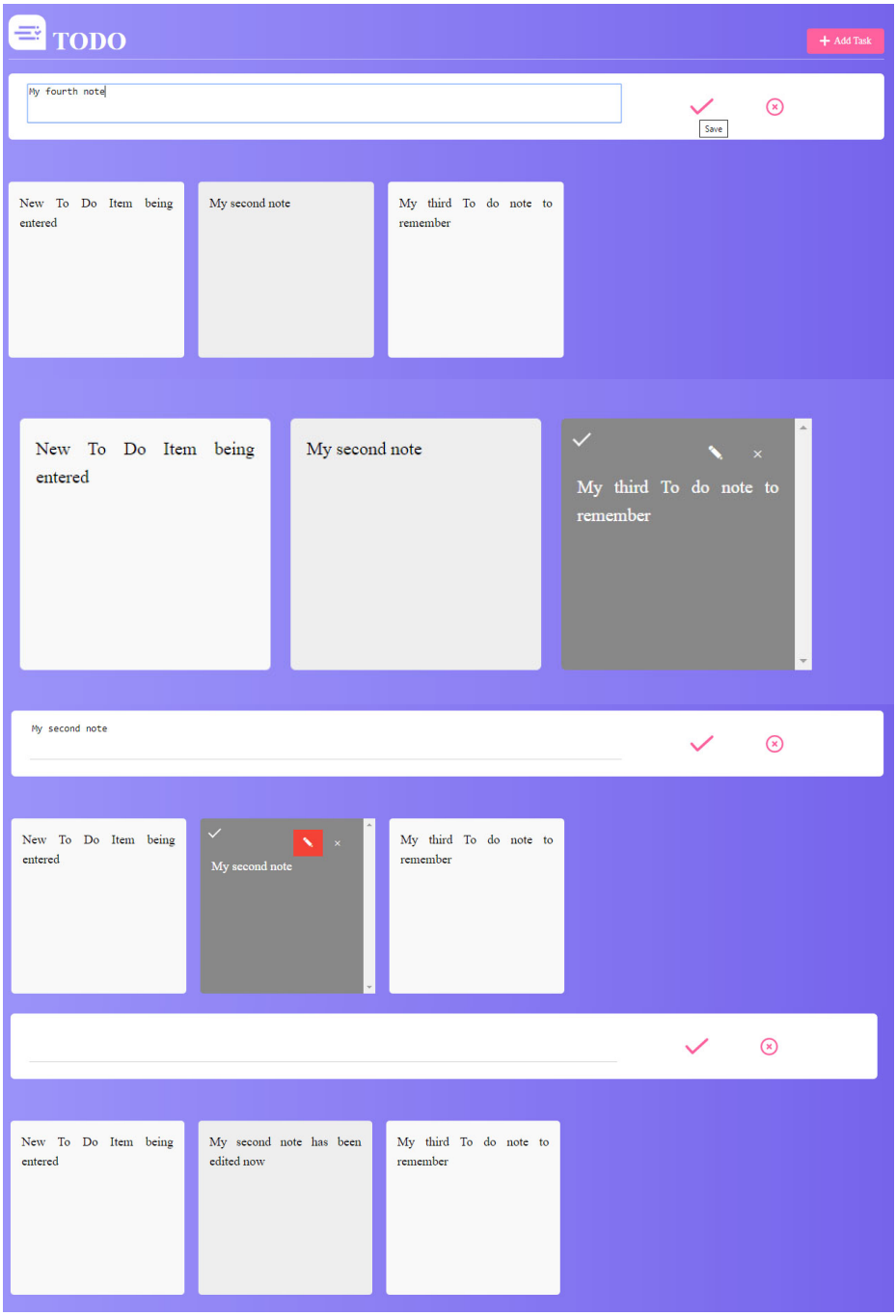


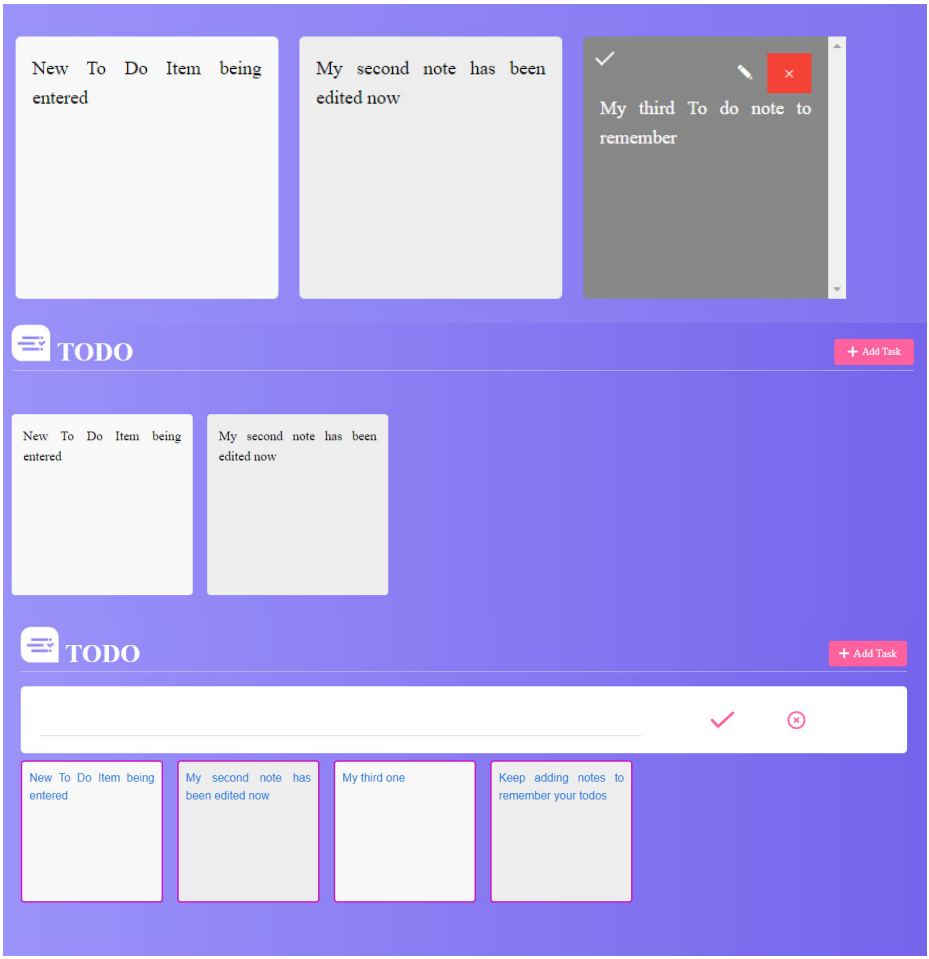


Organising always seems impossible until its DONE!

TODO


TODO
+ Add Task
Add your first todo by pressing "Add Task" button at the top right corner
TODO
+ Add Task
New To Do Item being entered
TODO
+ Add Task
New To Do Item being entered
TODO
+ Add Task
Second note being entered
Cancel
TODO
+ Add Task
Cancel


TODO
Add Task
My fourth note
Save
New To Do Item being entered
My second note
My third To do note to remember
New To Do Item being entered
My second note
My third To do note to remember
My second note
New To Do Item being entered
My second note
My third To do note to remember
New To Do Item being entered
My second note has been edited now
My third To do note to remember


New To Do Item being entered
My second note has been edited now
My third To do note to remember
TODO
+ Add Task
New To Do Item being entered
My second note has been edited now
TODO
+ Add Task
New To Do Item being entered
My second note has been edited now
My third one
Keep adding notes to remember your todos

CHAPTER 9

Building An Application with JavaScript

"You cannot score a goal when you are sitting on the bench. To do so, you have to dress up and enter the game."

– Israelmore Ayivor

So far, you learned about web content creation using HTML, styling the content using CSS, adding interactivity to the content using JavaScript, and the different aspects of JavaScript programming. Now is the time to

apply all that we have learned and see things come into action.

In this chapter, you will see how all this knowledge can be applied to see applications taking shape. You will be building a simple application to handle user entered notes.

Structure

- Planning the application
- Solution approach
- Building the application step-by-step

Objective

At the end of this chapter, you will understand how to approach an application development and steps to develop a simple JavaScript application.

Planning the application

The application which you will be building here is a single page application which enables users to add and keep notes. To keep it simple, since we do not have any backend services, the data will be maintained in the local storage, unlike a real application as that will involve introducing services. In order to further enhance the application, you can integrate it with a real service to maintain the notes data.

The main focus of this application is to make use of the knowledge of HTML, CSS, and JavaScript and build an application which will have the following two screens:

1. **An application launch screen:** In a real application, where we use third-party APIs, this will be close to the login screen. Here, it is used just to show navigation between the two screens.
2. A to do notes screen where the user can add, edit, and delete notes.

The preceding application will provide the following capabilities:

1. The application launch screen which will take the user to the notes screen using a mouse click.
2. The notes screen will provide the following capabilities around managing and handling notes:
    a. Add a new note. The note gets added to the existing list of notes.
    b. The notes will be displayed in a card list view. The newly added note will appear as the first note in the list.

c. The notes will be loaded from a static list initially saved in the local storage.
d. The new notes will be added to the same static list maintained in the local storage.
e. The user should be able to edit and delete existing notes by making use of the icons on the card.

3. All changes should be maintained in the local storage (as we are not making use of any external services).

In the planning phase, we will create a wireframe for the end application screen, which will help us put our thoughts together diagrammatically. The following image depicts the details for the current notes application and we will use the same to convert into a real application:

Figure 9.1: *Application wireframe*

Now with some visual idea of what we have to build, we will proceed with the solution.

Solution approach

The first step to build any application is to define its **user experience (UX)** very clearly. What do you mean by UX? How is it different from a user interface? In simple terms, a user interface includes how the screens of your application look, but user experience includes what kind of experience your application gives to the user. Defining user experience includes all of the following and much more:

1. How the screens or user interfaces look?
2. What are the different user interactions?
3. Are the different interactions easy and convenient to use?
4. Does it give a good feeling and presentation?
5. What is the overall experience of using the app?

User experience is a wide area of study on its own, but my idea was to introduce it to you at this point. You should always keep this in mind while building any solution and should explore this further to build applications with great UX.

Our current application is a simple notes app, so we will define the two screens and the different interactions as a part of this process.

In real applications, any data which needs to be persisted would be fetched from external services and any changes would be saved using the same services. You should explore the services further to become a full stack professional (also read about them in *Chapter 18, What next - for becoming*

a pro?).

This book only deals with the front end or the UI layer; hence, we will not be interacting with any external services, instead we will use the data by saving it in the local storage which will persist for the current session and will get refreshed for a new session. You will build the screens using HTML and CSS and then handle the different user interactions using JavaScript.

Building the application step-by-step

The overall application will comprise HTML, CSS, JS, and images files. The application folder structure is maintained in the following way:

- All HTML files will be created within the main project folder.
- The main project folder has the following subfolders:
    a. `resources`: This folder is used to store all application-related images in different formats like PNG, SVG, and so on. You can take a copy of this folder to get all the project resources as you start coding along.
    b. `js`: This folder is used to store all the JavaScript files for the project.
    c. `styles`: This folder will hold all the CSS files for the project.

Following a proper folder structure makes the code organized and maintainable.

Step 1 – Adding content to the launch page

The launch screen is only a placeholder screen to enter into the application.

We will first create the launch.html with the image content only.

The initial `launch.html` code looks like this:

```
<html>
<head>
<meta name="viewport" content="width=device-width,
initial-scale=1">
</head>
<body>
<div>
<div>
<imgsrc=".\resources\loginCapture.png" />
</div>
<div>
<div >
```

```
<p >Organising always seems impossible until it's DONE!
</p>
</div>
</div>
</div>
</body>
</html>
```

The corresponding page looks like the following screenshot:

Figure 9.2: *Launch screen content without styling*

The content looks very plain without any styling. Let's add some styling next and see how the page changes totally.

Step 2 – Styling the launch page

In order to apply styling, we will create a launch.css file where we will define all the related classes to be applied to `launch.html`.

The following CSS code is added to style the image to the left-hand side of the page with the content to the right and a **`ToDo`** launch button, which when clicked loads the main application:

```
/* loginDiv is main div in the login page */
.loginDiv{
 display: flex;
 margin: 86px;
 height: 75%;
}
/* These css styles are applied for the left side image
in the login page */
.notepadCls{
 border-top-right-radius: 235px;
 border-bottom-right-radius: 235px;
 background-color: #FFFFFF;
 width: 70%;
 margin-left: 70px;
 z-index: 1;
}
.todoImage{
 width: 60%;
 height: 80%;
 margin-top: 60px;
}

/* These css styles are applied for the right side part
of the login page */
```

```
.todoOuterCls{
 position: absolute;
 height: 73%;
 left: 46%;
 background-color: #7563eb;
 border-radius: 10px;
 width: 40%;
}

.todoDivCls{
 background-color: #ff619e;
 border-radius: 10px;
 margin-top: 50px;
 margin-left: 40px;
 width: 80%;
 height: 80%;
 z-index: 2;
 position: relative;
}
.todoInnerDivCls{
 padding: 10% 0 0 0%;
}
/*Include the color and font family to the title*/
.loginTitle {
 font-size: 2vw;
 width: 30vw;
color: #FFFFFF;
 font-family: Montserrat sans-serif;
 text-align: center;
}
/*Include height and padding for the image*/
.loginTodoCls {
 height: 60px;
 padding: 16% 0 0 30%;
}
```

The final `launch.html` is as follows:

```
<html>
<head>
<meta name="viewport" content="width=device-width,
initial-scale=1">
</head>
<body>
<div class="loginDiv">
<!-- This div is for Todo image to show on the left
```

```
side of screen -->
<div class="notepadCls">
<imgsrc=".\resources\loginCapture.png"
class="todoImage"/>
</div>
<!-- This div is for design of Todo login part on the
right side of screen -->
<div class="todoOuterCls">
<div class="todoDivCls">
<!-- This div is inner div of login part to align the
paragraph and image on the right side of screen-->
<div class="todoInnerDivCls">
<p class="loginTitle">Organising always seems
impossible until its DONE!</p>
<imgsrc = ".\resources\SidemenuLogo.svg"
class="loginTodoCls" onClick="loadMainPage()" />
</div>
</div>
</div>
</div>
<script type="text/javascript" src=".\js\launch.js"></
script>
<link rel="stylesheet" type="text/css" href=".\styles
\launch.css" >
</body>
</html>
```

The styled launch page looks like this:

Figure 9.3: *Launch screen content with styling*

The page looks much better now. Right! That's the power of CSS!

Step 3 – Logic for the launch page

The only logic for the launch page is that when you click on the `ToDo` button, it should navigate to the main application page.

Create a `launch.js` file inside the `js` folder with the following code for the `loadMainPage` function which is the handler for the `onclick` action on the image:

```
function loadMainPage(){
window.location.href = "./index.html";
}
```

This will load the `index.html` page which is the main application page.

Step 4 – Adding responsiveness to the launch page

Let's include some responsiveness in the launch page to give it the final finish by including media queries in launch.css for some screen size thresholds as follows:

```
@media screen and (max-width: 700px) {
.loginDiv {
    width: 85%;
    margin: 86px 10px 86px 35px;
 }

.notepadCls {
    width: 45%;
 }

.loginTodoCls {
      padding: 12% 0 0 12%;
      height: 35px;
 }
}
@media screen and (max-width: 650px) {
.todoDivCls {
      margin-left: 20px;
 }
}

@media screen and (max-width: 400px) {
.loginTodoCls {
      height: 35px;
 }
.todoDivCls {
      margin-left: 20px;
 }
.loginDiv {
      margin: 86px 10px 86px 35px;
 }
}
```

Now, you are done with the launch page which will take us to the main application page.

Step 5 – Content and styling for the application page

Now, we will work on the main application page for handling notes.

Let's start with the `index.html` file and first add and style the header content as follows:

```
<!DOCTYPE html>
<html>
<head>
<meta name="viewport" content="width=device-width,
initial-scale=1">
</head>
<div id="headerDIV" class="header">
<!-- this div is for main logo of top left corner -->
<div>
<imgsrc = ".\resources\SidemenuLogo.svg"
class="mainLogo" alt="ToDoLogo">
</div>
<!--Ths div tag is for add task button at top right
corner -->
<div class="headNavButtonsContainer">
<div class="addTask">
<a class="linkTag" onClick ="showInputText()" >
<imgsrc = "
xcYVwJLmxFXAl9q0CGIYEHjahF3dSxxcWDxHOMiFTjqos36ZIbiZqt5MJF
SyZgB0oLOgAnqFIh8SWNdAAAAABJRU5ErkJggg==" class="white
headNavIcons">
<p class="btntext">Add Task</p>
</a>
</div>
</div>
</div >
<div class="noDataFound">Add your first todo by
pressing “Add Task” button at the top right corner</
div>
</body>
</html>
```

Next, add some body level styling in `Main.css` as follows:

```
/*Include the background color to the body*/
body {
 margin: 0;
 font-family: Montserrat;
 -webkit-font-smoothing: antialiased;
 -moz-osx-font-smoothing: grayscale;
 background-image: linear-
 gradient(90deg,#9c94f9,#7563eb);
}
/* Include the padding and border in an element's total
```

```
width and height */
* {
 box-sizing: border-box;
}
```

Include the header styling in Main.css to style the header text, logo, and buttons as shown in the following code:

```
/* Style the header */
.header {
 margin: 0 60px 0 60px;
 height: 14vh;
 display: -webkit-flex;
 -webkit-align-items: flex-end;
 align-items: flex-end;
 border-bottom: 1px solid;
 border-color: hsla(0,0%,100%,.5);
color: #fff;
 padding-top: 25px;
}
/*Include width and height to the main logo*/
.mainLogo {
 height: 9.7vh;
 width: 175px;
}
/* styles related to the top right corner button*/
.headNavButtonsContainer {
 -webkit-justify-content: flex-end;
 justify-content: flex-end;
 margin-bottom: .6%;
 width: 87vw;
}
.headNavButtonsContainer, .linkTag {
 display: flex;
 -webkit-align-items: center;
 align-items: center;
}
.addTask {
 height: 3vh;
 cursor: pointer;
 background-color: #ff619e;
 border-radius: 4px;
 padding: 18px 0;
}
.addTask, .addTaskBtnIconsContainer {
```

```
 -webkit-align-items: center;
 align-items: center;
 display: flex;
}
.linkTag {
 width: 110px;
 height: 3vh;
 -webkit-justify-content: center;
 justify-content: center;
}
.headNavIcons {
 width: 15px;
}
.btntext {
 padding-left: .4vw;
 font-size: 14px;
}
```

With no notes present, when you land on the application for the first time, you will get the following view with the header:

Figure 9.4: *ToDo Notes application landing page when no notes present*

Next, we will include the body part. The notes will be dynamically added when the user enters data or gets preloaded from the local storage which will be done in JavaScript.

The following section will render the `textarea` to enter the note along with two buttons to add the note and clear the note text which is appropriately handled by JavaScript functions: `createTodo()` and `clearText()`:

```
<!-- This div tag is to show textarea, save and delete
icon -->
<div class="addTaskContainer" id="addtaskcontainerid">
<div id="ItemContainer" class="itemcontainer">
<!-- This div tag is to show textarea -->
<div class="taskInputFieldContainer">
<textarea class="addTaskInputField" id="todoinputfield"
type="text" required=""></textarea>
</div>
<div class="addTaskEditBtnContainer">
<!-- This div tag is to show save icon -->
<div class="addTaskBtnIconsContainer" title="Save" >
<imgsrc = ".\resources\ConfirmTaskSVG.svg"
class="saveTaskIcon" onClick= "createTodo()">
</div>
```

```
<!-- This div tag is to show delete icon -->
<div id ="cancelBtnId" class="deleteIconsContainer"
title="Cancel">
<imgsrc = ".\resources\Delete.svg" onClick =
"clearText()">
</div>
</div>
</div>
</div>
```

Next, we need to style the cards of tasks and the task entry input box as shown in the following code:

```
/*Styles related to the input text area */
.addTaskContainer {
color: #fff;
 display: none;
 margin: 20px 60px 0 60px;
}
.itemcontainer {
 border-radius: 6px;
 display: -webkit-flex;
 display: flex;
 -webkit-flex-wrap: nowrap;
 flex-wrap: nowrap;
 -webkit-justify-content: start;
 justify-content: start;
 background-color: #fff;
}
/*Styles related to the input text area */
.taskInputFieldContainer {
 width: 75%;
 padding: 1.2% 5% 1.6% 2.2%;
}
.addTaskInputField {
 resize: none;
 width: 100%;
 height: 55px;
 border: none;
 border-bottom: 1px solid #d3d3d3;
 overflow-wrap: break-word;
 background-color: initial;
 flex-wrap: wrap;
 -webkit-flex-wrap: wrap;
 font-size: 14px;
```

```
}
```

We can add some styling to the icons to handle the save and delete actions on the `todo` text input area as follows:

```
/*Styles related to the save icon*/
.addTaskEditBtnContainer {
 display: -webkit-flex;
 display: flex;
 width: 25%;
}
.addTaskBtnIconsContainer {
 width: 33.3%;
 -webkit-justify-content: center;
 justify-content: center;
}
.saveTaskIcon {
 height: 25%;
}
/*Styles related to the delete icon*/
.deleteIconsContainer {
 width: 33.3%;
 -webkit-justify-content: center;
 justify-content: center;
 display: none;
}
```

The text input for TODO looks like the following screenshot:

Figure 9.5: *ToDo notes screen with Note being entered*

The preceding screen should appear when you click on the `AddTask` button and also handle the saved content of the notes in the local storage. All this logic can be handled only using JavaScript.

Step 6 – Content and styling for notes list in the application page

The content for the notes will be added dynamically by the JavaScript logic as it will be based on the data created by the user, which we will see in the next step. In this step, let's include some styling for the list.

The following styling can be included in `main.css` for the list items and the card layout for the note:

```
/* Style the list items */
ul li {
 cursor: pointer;
```

```
 position: relative;
 padding: 12px 8px 12px 40px;
 list-style-type: none;
 background: #eee;
 font-size: 18px;
 transition: 0.2s;

 /* make the list items unselectable */
 -webkit-user-select: none;
 -moz-user-select: none;
 -ms-user-select: none;
 user-select: none;
}

/* Set all odd list items to a different color (zebra-
stripes) */
ul li:nth-child(odd) {
 background: #f9f9f9;
}

/* Darker background-color on hover */
ul li:hover {
 background: #ddd;
}
/* When clicked on, add a background color and strike
out text */
ul li.checked {
 background: #888;
color: #fff;
}

/* Add a "checked" mark when clicked on */
ul li.checked::before {
 content: '';
 position: absolute;
 border-color: #fff;
 border-style: solid;
 border-width: 0 2px2px 0;
 top: 10px;
 left: 16px;
 transform: rotate(45deg);
 height: 15px;
 width: 7px;
}

/*Styles related to the list item */
```

```
.card {
 padding: 16px;
 width: 250px;
 height: 250px;
 margin-right: 20px;
 margin-bottom: 10px;
 border-radius: 5px;
 overflow: hidden;
 overflow-wrap: break-word;
 line-height: 1.6;
}
.card:hover{
 overflow-y: scroll !important;
}

/* styles related to the list items */
.listcolumn {
 display: flex;
 flex-wrap: wrap;
 margin: 10px60px60px60px;
}
```

With this styling associated with the list items, we will now look at the logic to handle the notes.

Step 7 – Logic for the application page

Now, let's look at the JavaScript to handle the button click functionality for adding the task button, `createTodo()` (to add a note), and `clearText()` (to clear the user entered text). We will start shaping up main.js with all the main application-related JavaScript logic.

Let's start with handling of the `addTask` button to show the input text area when you click on the `Add task` button and handling the show and hide behavior of the Delete icon as follows:

```
consttextfieldInput =
document.getElementById("todoinputfield");
constinputTextfield =
document.getElementById('addtaskcontainerid');
const list = document.querySelector('ul');
var editClicked = false;
var listElement = '';
/**
 * Added keyup listener to show and hide of delete icon
 if text is available or not
 */
```

```
textfieldInput.addEventListener("keyup", function() {
constnameInput = textfieldInput.value;
 if (nameInput != "") {
document.getElementById('cancelBtnId').className =
"addTaskBtnIconsContainer";
 } else {
document.getElementById('cancelBtnId').className =
"deleteIconsContainer";
 }
});
/**
 * This method is to show the input textfield on click
 of Add Task button
 */
function showInputText(){
 var cardView =
 document.getElementById('todoitemslist');
this.noDataFound();
inputTextfield.style.display = 'block';
cardView.style.margin = '60px';
}
/**
 * noDataFound method is to show some text when list
 items are empty
 */
function noDataFound(){
 var defaultText =
 document.getElementsByClassName('noDataFound')[0];
defaultText.style.display = "none";
}
```

The **clearText** function will get the text reference using getElementById and reset the value to ‘’;:

```
function clearText() {
let inputValue =
document.getElementById("myInput").value;
 if (inputValue === '') {
alert("No text to clear!");
 } else {
document.getElementById("myInput").value = '';
 }
}
```

Next, let’s discuss the set of functions to handle the create and delete buttons on the input text area.

The `createTodo` function is called when the user enters some note and wants to add it to the card list. This means a new list element of the card view will be created with the current details and appended to the DOM:

```
/**
 * This method is to add the entered text on input
 textfield
 */
function createTodo() {
  if(editClicked) {
this.editedData();
editClicked = false;
return ;
 }

 var inputValue =
 document.getElementById("todoinputfield").value;
 if (inputValue === '') {
return ;
 }
 var li = document.createElement("li"),
randomId = Math.floor(Date.now() / 1000), //
Math.random().toString(36).substr(2, 9);
divElement = '';
this.addDeleteAndEditBtn(li);
divElement = document.createElement("DIV");
divElement.innerHTML = inputValue;
divElement.className = "cardViewInput";
li.id = randomId;
li.appendChild(divElement);

document.getElementById("todoitemslist").appendChild(li);
li.className = "card";

if(inputValue.length !=0) {
   var itemsArray = localStorage.getItem('cardItems')?
   JSON.parse(localStorage.getItem('cardItems')) : [];
itemsArray.push({
     text: inputValue,
     id: randomId
   });
localStorage.setItem('cardItems',
JSON.stringify(itemsArray));
 }
document.getElementById("todoinputfield").value = "";
}
```

Now, since the page has to show the content dynamically, the logic of adding the card with the new node will be handled in the JS function as shown in the preceding code.

When you click on the tick (save) icon, the data will be stored in the local storage and displayed in a list as shown in the following screenshot:

Figure 9.6: *The ToDo notes list is added when you click on the tick icon*

The `clearText` function is used the delete icon on the input text area, which will clear the text if the text was entered and hide the input text area if no text was entered as shown in the following code:

```
/**
 * This method is to clear the entered text on input
 textfield
 */
function clearText() {
if(textfieldInput.value === '') {
document.getElementById('cancelBtnId').className =
"deleteIconsContainer";
inputTextfield.style.display = 'none';
  }
document.getElementById("todoinputfield").value = '';
}
```

This handles clearing of the text as well as hiding the input text area:

Figure 9.7: *The delete icon*

When you click on the Delete icon (above), the text will be cleared as shown in the following screenshot:

Figure 9.8: *The delete icon clears the text entered*

When you click on the second time, the complete text area will disappear as shown in *Figure 9.5*.

As more notes are added using the tick (Save) icon, they appear in the cards view as shown in the following screen:

Figure 9.9: *The delete icon clears the text entered*

Now, we also need to add some logic on the card level to be able to delete or edit the content already entered.

The following click event listener is included to listen to the click event and show the icons to make changes to the card data:

```
/**
 * Add a "checked" symbol when clicking on a list item
 * show the edit and delete icons when click on the
```

```
 list item
 * handle the cases when click on list item and div
 (text in the list item)
 */

list.addEventListener('click', function(ev) {
 // get the list items by query the dom
 var checkedList =
 document.querySelectorAll('.checked'),
tagName = ev.target.tagName,
checkedTargetValue = tagName === 'LI'?
event.target.classList.contains('checked') :
ev.target.parentElement.classList.contains('checked');

 // return empty when already select a card
if(checkedList.length>=1 && !checkedTargetValue) {
return ;
 }
 if (tagName === 'LI') {
ev.target.classList.toggle('checked');
 } else if(tagName === 'DIV') {
ev.target.parentElement.classList.toggle('checked');
 }
checkedTargetValue = tagName === 'LI'?
event.target.classList.contains('checked') :
ev.target.parentElement.classList.contains('checked');
 // added the condition based on the checkedTargetValue
 to show or hide the edit or delete icons
 if(checkedTargetValue) {
if(tagName === 'LI') {
ev.target.children[0].style.display = 'flex';
ev.target.getElementsByClassName('close')[0].onclick =
function(event){
window.onDeleteIconClick(ev.target);
event.stopPropagation();
      }
ev.target.getElementsByClassName('edit')[0].onclick =
function(event){
window.onEditIconClick(ev.target);
event.stopPropagation();
      }
     } else if(tagName === 'DIV') {
ev.target.previousElementSibling.style.display =
'flex';
ev.target.parentElement.getElementsByClassName('close')
```

```
[0].onclick = function(event){
window.onDeleteIconClick(ev.target.parentElement);
event.stopPropagation();
      }
ev.target.parentElement.getElementsByClassName('edit')
[0].onclick = function(event){
window.onEditIconClick(ev.target.parentElement);
event.stopPropagation();
      }
    }
  } else if(!checkedTargetValue) {
if(tagName === 'LI') {
ev.target.children[0].style.display = 'none';
  } else if(tagName === 'DIV') {
    // Handle when click on the icon's div to unselect
    the card
    if(ev.target.previousElementSibling) {
ev.target.previousElementSibling.style.display =
'none';
    } else {
ev.target.style.display = 'none';
    }
  }
  // Clear the text of textarea when deselecting the
  card
editClicked = false;
document.getElementById("todoinputfield").value = '';
  }
}, false);
```

The icons for delete and edit on the card are attached on the card using the following function logic:

```
/**
 * addDeleteAndEditBtn is used to create element for
 edit and delete buttons
 */
function addDeleteAndEditBtn(li) {
 var span = document.createElement("SPAN");
 var txt = document.createTextNode("\u00D7");
 var div = document.createElement("DIV");
span.className = "close";
div.className = "iconCls";
span.appendChild(txt);

 var editSpan = document.createElement("SPAN");
```

```
 var edittxt = document.createTextNode("\u270E");
editSpan.className = "edit";
editSpan.appendChild(edittxt);

div.appendChild(span);
div.appendChild(editSpan);
li.appendChild(div);
}
```

This will include the two icons on the card to handle actions to manipulate the card data as shown when you select one of the cards by clicking as shown in the following screenshot:

Figure 9.10: *The third card is selected*

To handle the icons added for edit and delete, handler functions are associated with the click actions.

There are two helper functions used to attach the delete and edit operations to the card view. The logic is defined as a separate function so it can be reused whenever we need to render a card with these two operations.

Let's look at the two functions:

This will handle the logic for the click action of the delete icon; the particular card represented by the ID uniquely will be deleted from the card items list as well as will be updated in `localStorage`:

```
/**
 * @param li
 * @method onDeleteIconClick
 * This method is for handle the delete of selected
 list item
 */
function onDeleteIconClick(li) {
   var localStorageValues = [],
       index;

localStorageValues =
JSON.parse(localStorage.getItem('cardItems'));
for(var k=0; k <localStorageValues.length; k++) {
     if(localStorageValues[k].id == li["id"]) {
       index = k;
localStorageValues.splice(index, 1);
li.style.display = "none";
localStorage.setItem('cardItems',
JSON.stringify(localStorageValues));
       break;
```

```
      }
    }
if(localStorageValues.length == 0) {
      var defaultText =
      document.getElementsByClassName('noDataFound')[0];
inputTextfield.style.display = "none";
defaultText.style.display = "inline-block";
    }
}
```

Similarly, to handle the edit icon in order to update the edited text in the corresponding card, the following handler will be invoked:

```
/**
 *
 * @param li
 * @method onEditIconClick
 * This method is for add text to the textarea when
 click on the edit button
 */
function onEditIconClick(li) {
   var innerText = li.childNodes[1].innerHTML;
inputTextfield.style.display = 'block';
document.getElementById("todoinputfield").value =
innerText;
document.getElementById('cancelBtnId').className =
"addTaskBtnIconsContainer";
editClicked = true;
listElement = li;
}
```

Also, when you click on the edit icon, the selected note should show up in the input text area to be edited accordingly. This is handled by the following function to handle editing in a smooth manner as shown in the following code:

```
/**
 * @method editedData
 * This method is to add the edited data to the list
 item when clicked on save
 */
function editedData() {
 var editStorageValues = [],
   index,
editedText =
document.getElementById("todoinputfield").value;
```

```
listElement.childNodes[1].innerHTML = editedText;
editStorageValues =
JSON.parse(localStorage.getItem('cardItems'));
for(var m=0; m <editStorageValues.length; m++) {
 if(editStorageValues[m].id == listElement["id"]) {
    index = m;
editStorageValues[index].text = editedText;
localStorage.setItem('cardItems',
JSON.stringify(editStorageValues));
    break;
   }
 }
document.getElementById("todoinputfield").value = '';
listElement.classList.toggle('checked');
listElement.children[0].style.display = 'none';
}
```

Now, when you select the edit icon, the selected note text appears in the input text area to be edited as shown in the following screenshot:

Figure 9.11: *The edit icon in action*

The edited text can be saved using the tick icon to reflect on the notes as follows:

Figure 9.12: *The second note has been edited*

In the same way, using the cross (delete) icon on the note, the note card can be deleted from the view and the local storage now as shown after deleting the third note:

Figure 9.13: *The delete button in action*

The third note will be removed from the view and local storage after clicking on the button above as shown in the following screenshot:

Figure 9.14: *The third note is deleted*

Now, the notes list is being successfully rendered and the logic to handle the actions to manipulate the existing data and add new data in place.

In order to ensure, we re-render the same data; if the user reloads or refreshes the page, we need to include another additional logic to rebuild the list of cards from the local storage.

The following code is included in index.html to handle refresh of page by reloading the data in localstorage:

```
<body onload="restoreData()">
```

The handler JavaScript code will re-render the entire data list in the card

view with the required delete and edit icons:

```
/**
 * This method is to display all the existing card
 views when refresh the page
 */
function restoreData() {
 if (typeof(Storage) !== "undefined") {
 var todoItems
 =JSON.parse(localStorage.getItem('cardItems'));
if(todoItems&&todoItems .length !=0) {
this.noDataFound();
 }
for(var key in todoItems ) {
    var li = document.createElement("li"),
divElement = '';
this.addDeleteAndEditBtn(li);
divElement = document.createElement("DIV");
divElement.className = "cardViewInput";
divElement.innerHTML = todoItems [key].text;
li.id = todoItems[key].id;
li.appendChild(divElement);
document.getElementById("todoitemslist").appendChild(li);
li.className = "card";
   }
 }
}
```

With this logic in place, the complete notes application is functional to handle any todo notes list.

If you want to change some styling, feel free to explore. I thought the note cards look too big so I changed the dimensions and added some border. This is how my final application looks like:

Figure 9.15: *The Notes application*

The entire code for this application can be found in the code bundle accompanying this book.

Conclusion

In this chapter, we built a simple notes application using the knowledge of HTML, CSS and JavaScript. Putting our knowledge to practice will give us the required understanding and comfort while coding. We learned how you should plan your application by considering the different aspects of requirements, including the user interfaces, the user experience, and the

data elements for your application. After having planned the different elements of development, we proceeded with the solution approach. Later, we built the application step-by-step starting with adding content using HTML, styling the content using CSS, and lastly adding logic and interactiveness using JavaScript. We should practice building more such applications to get a hold of these three building blocks of web development, the knowledge of which is useful irrespective of whichever framework we end up using in future.

Also, as we get comfortable, we will start working on HTML, CSS, and JS all hand-in-hand.

In the next chapter, we will learn the skills of debugging JavaScript applications and the tools which we use for debugging during development.

Questions

1. Which of the following indicate user experience for an application?

    A. How the screens or user interfaces look like?

    B. What are the different user interactions?

    C. Are the different interactions easy and convenient to use?

    D. All of the above

    Answer: D

2. The project folder should comprise the following folders mandatorily?

    A. Styles or CSS for the CSS files

    B. Js for the JavaScript files

    C. Resources or assets for the static files and images

    D. None of the above are mandatory but are required to maintain a clean and understandable code structure

    Answer: D

3. Which of these lines of code in JS hides a div container?

    A. `span.appendChild(txt);`

    B. `document.getElementById("myUL").appendChild(li);`

    C. `defaultText.style.display = "none";`

    D. None of the above

    Answer: C

4. The following code snippet can do the following

```
document.createTextNode("\u270E");
```

A. Include an edit icon

B. Include a delete icon

C. Include a close icon

D. None of the above

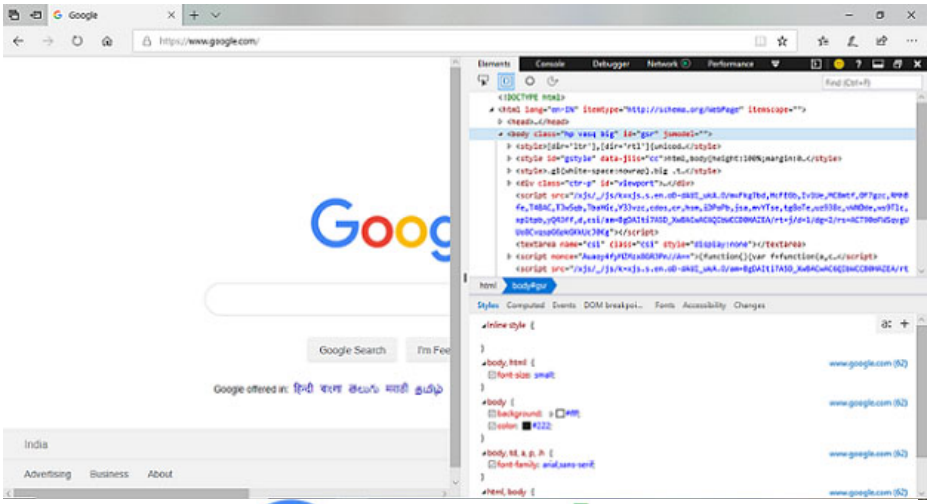
Answer: A

5. In real applications, any data which needs to be displayed and persisted would be fetched from external services and any changes would be saved using the same services.

A. TRUE

B. FALSE

Answer: A.



Google

Select all
Print
Read aloud
Paste
View source
Inspect element

Google Search I'm Feeling Lucky

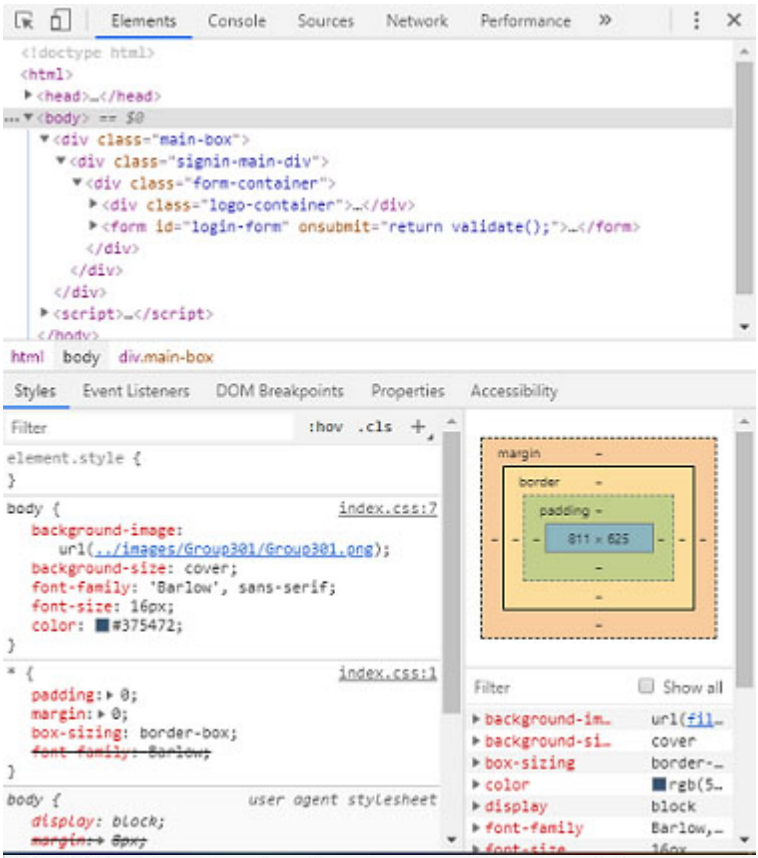
Google offered in: हिन्दी বাংলা తెలుగు ಕನ್ನಡ മലയാളം ਪੰਜਾਬੀ

Elements Console Sources Network Performance 3

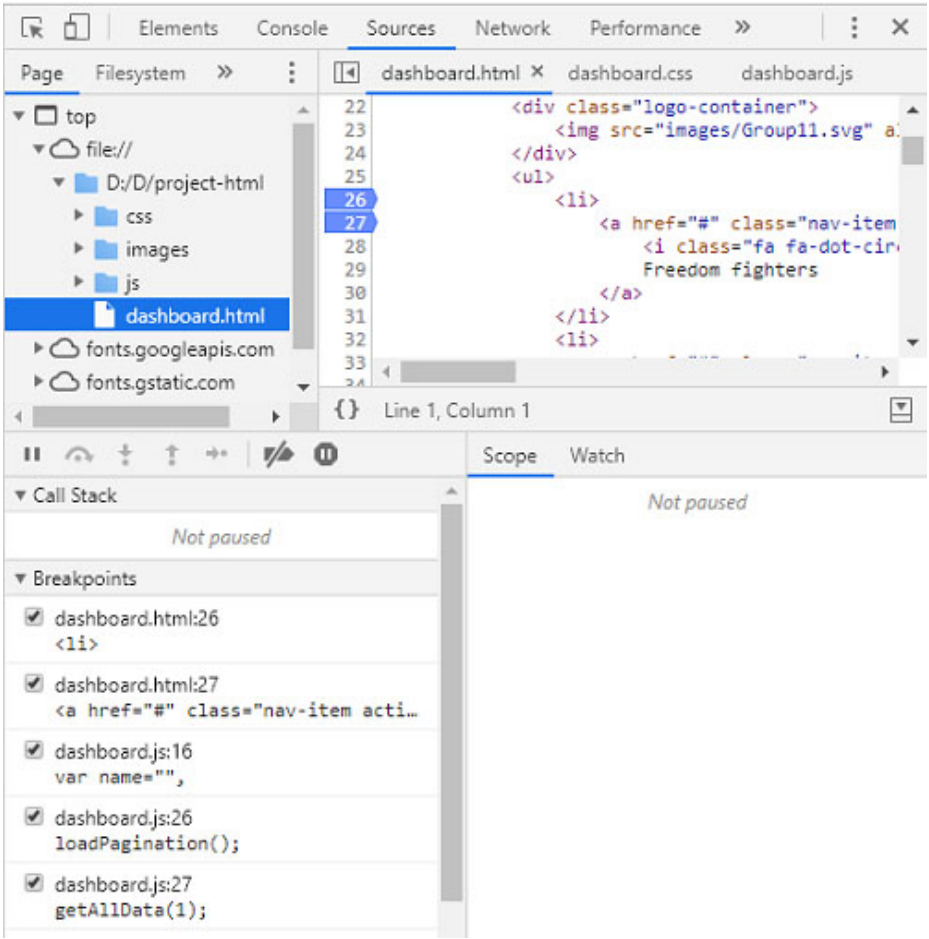
```
<div id="custom-bg" style="opacity: 0;"></div>
<!-- Container for the OneGoogleBar HTML. -->
<div id="one-google" class>…</div>
<script type="text/javascript">…</script>
<div id="ntp-contents" class="default-theme">
  <div id="logo">…</div> == $0
  <div id="fakebox-container">…</div>
  <!-- TODO(crbug/944624): Remove wrapper afte
  <div id="user-content-wrapper">…</div>
  <div id="attribution" style="display: none;"
  <div id="error-notice-container">…</div>
  <div id="edit-bg" tabindex="0">…</div>
```

Dock side

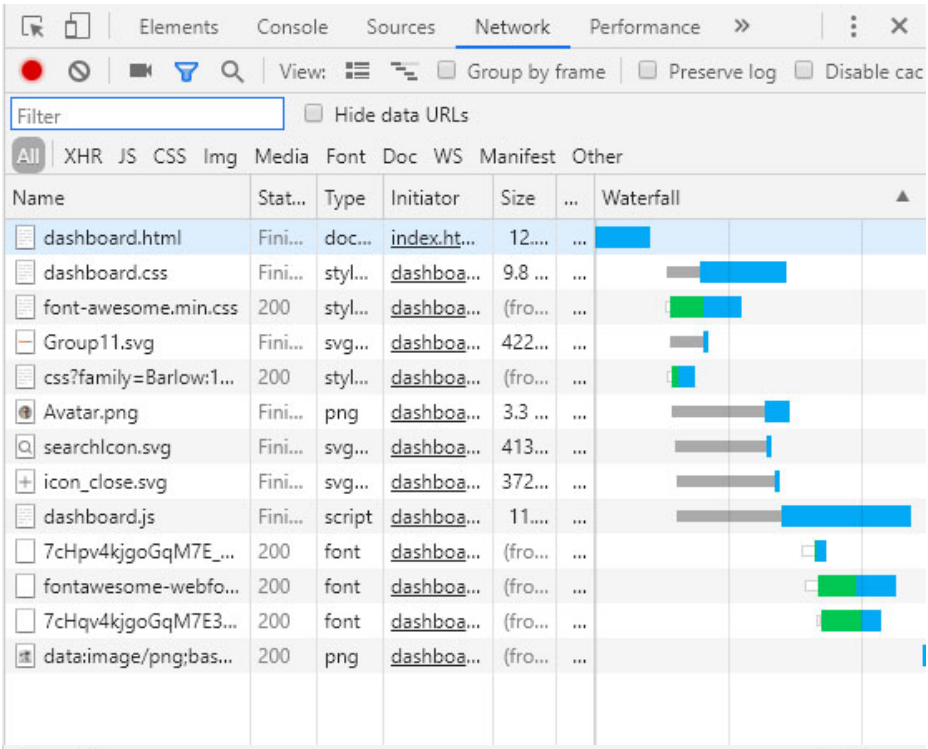
Show console drawer Esc
Search Ctrl + Shift + F
Run command Ctrl + Shift + P
Open file Ctrl + P
More tools
Shortcuts
Settings F1
Help

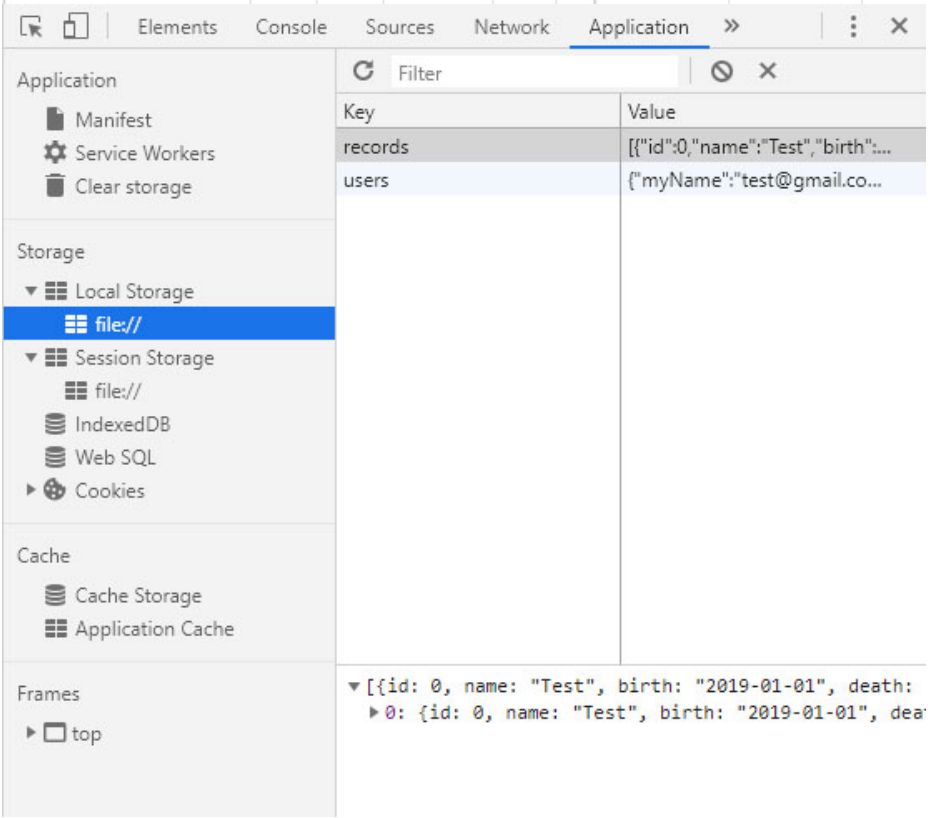

Elements
Console
Sources
Network
Performance
<!doctype html>
<html>
<head>…</head>
<body> == $0

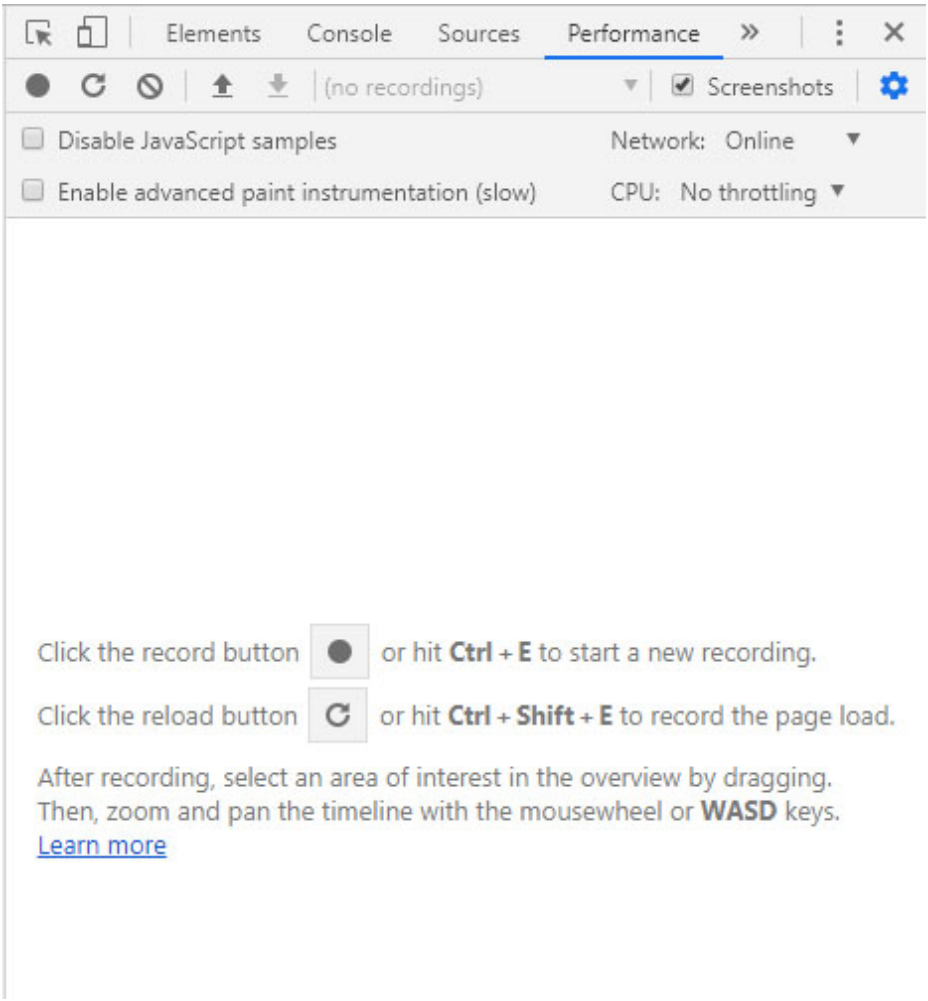
…
<form id="login-form" onsubmit="return validate();">…</form>
<script>…</script>
html
body
div.main-box
Styles
Event Listeners
DOM Breakpoints
Properties
Accessibility
Filter
:hov .cls
element.style {
}
body {
index.css:7
background-image:
url(../images/Group301/Group301.png);
background-size: cover;
font-family: 'Barlow', sans-serif;
font-size: 16px;
color: #375472;
}
* {
index.css:1
padding: 0;
margin: 0;
box-sizing: border-box;
font-family: Barlow;
}
body {
user agent stylesheet
display: block;
margin: 8px;
margin
border
padding
811 × 625
Filter
Show all
background-im…
url(fil…
background-si…
cover
box-sizing
border-…
color
rgb(5…
display
block
font-family
Barlow,…
Elements
Console
Sources
Network
Performance
top
Filter
Default levels
10+4
14
arr1=[1,2,3,4]
(4) [1, 2, 3, 4]
arr2=[4,5,6,7,8]
(5) [4, 5, 6, 7, 8]
arr3=arr1.concat(arr2)
(9) [1, 2, 3, 4, 4, 5, 6, 7, 8]

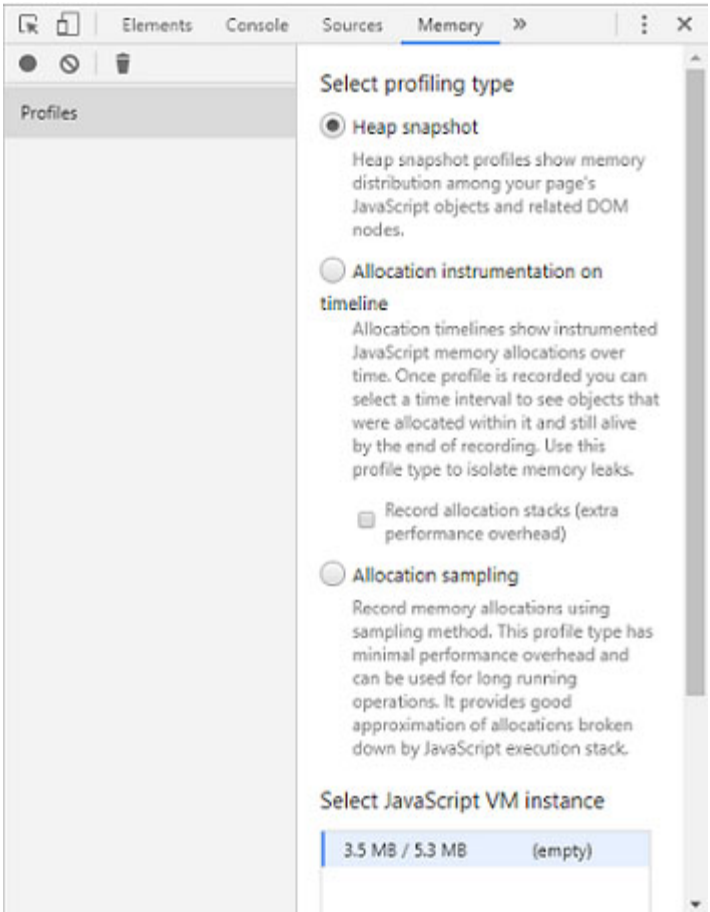

Elements
Console
Sources
Network
Performance
Page
Filesystem
dashboard.html
dashboard.css
dashboard.js
top
file://
D:/D/project-html
css
images
js
dashboard.html
fonts.googleapis.com
fonts.gstatic.com
22
23 <img src="images/Group11.svg" a
24
25
26
27 <a href="#" class="nav-item
28 <i class="fa fa-dot-cir
29 Freedom fighters
30
31
32
Line 1, Column 1
Scope
Watch
Not paused
Call Stack
Not paused
Breakpoints
dashboard.html:26

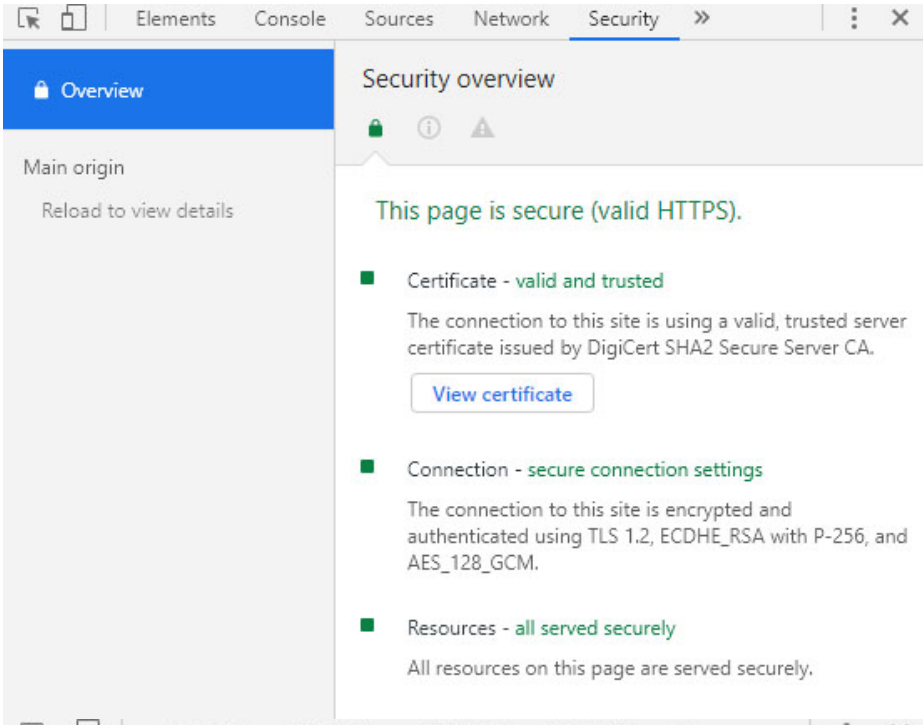
dashboard.html:27
<a href="#" class="nav-item acti…
dashboard.js:16
var name="",
dashboard.js:26
loadPagination();
dashboard.js:27
getAllData(1);

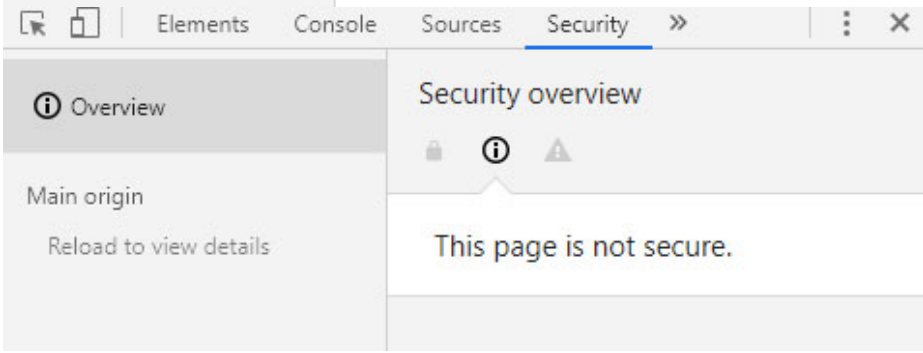

Elements
Console
Sources
Network
Performance
View:
Group by frame
Preserve log
Disable cac
Filter
Hide data URLs
All XHR JS CSS Img Media Font Doc WS Manifest Other
Name
Stat...
Type
Initiator
Size
...
Waterfall
dashboard.html Fini... doc... index.ht... 12.... ...
dashboard.css Fini... styl... dashboa... 9.8 ... ...
font-awesome.min.css 200 styl... dashboa... (fro... ...
Group11.svg Fini... svg... dashboa... 422... ...
css?family=Barlow:1... 200 styl... dashboa... (fro... ...
Avatar.png Fini... png dashboa... 3.3 ... ...
searchIcon.svg Fini... svg... dashboa... 413... ...
icon_close.svg Fini... svg... dashboa... 372... ...
dashboard.js Fini... script dashboa... 11.... ...
7cHpv4kjgoGqM7E_... 200 font dashboa... (fro... ...
fontawesome-webfo... 200 font dashboa... (fro... ...
7cHqv4kjgoGqM7E3... 200 font dashboa... (fro... ...
data:image/png;bas... 200 png dashboa... (fro... ...

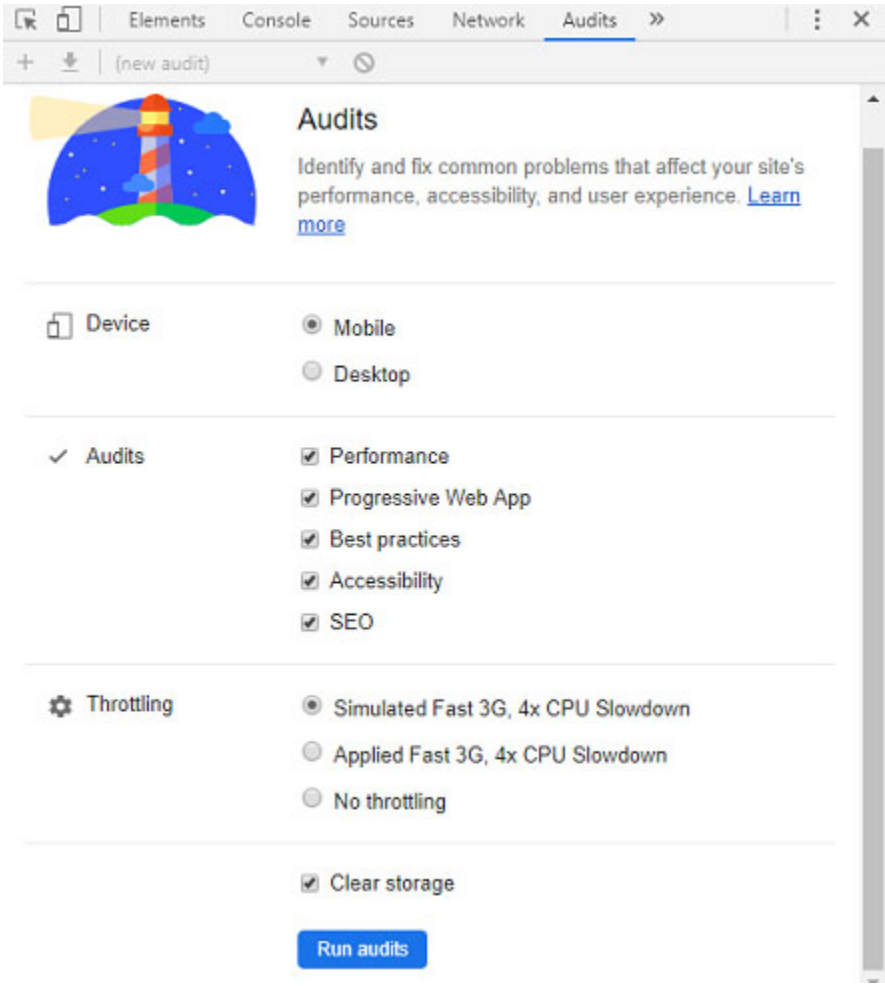

Elements
Console
Sources
Network
Application
Application
Manifest
Service Workers
Clear storage
Storage
Local Storage
file://
Session Storage
file://
IndexedDB
Web SQL
Cookies
Cache
Cache Storage
Application Cache
Frames
top
Filter
Key
Value
records
[{"id":0,"name":"Test","birth":...
users
{"myName":"test@gmail.co...
[{id: 0, name: "Test", birth: "2019-01-01", death:
0: {id: 0, name: "Test", birth: "2019-01-01", dea

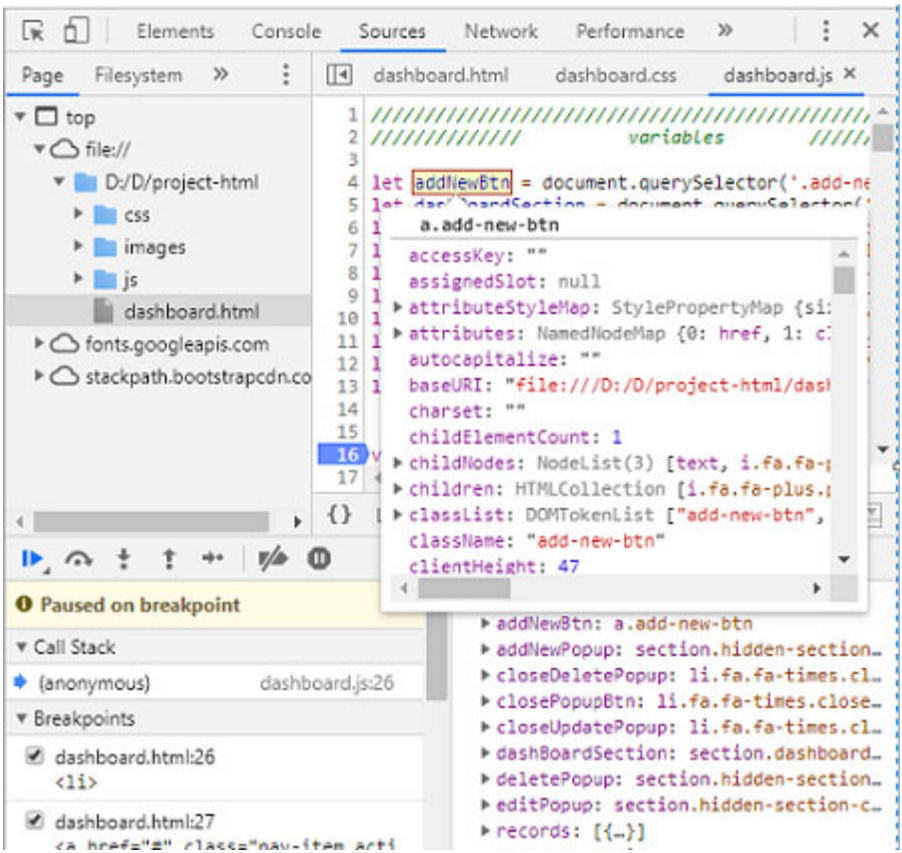

Elements
Console
Sources
Performance
(no recordings)
Screenshots
Disable JavaScript samples
Network: Online
Enable advanced paint instrumentation (slow)
CPU: No throttling
Click the record button or hit Ctrl + E to start a new recording.
Click the reload button or hit Ctrl + Shift + E to record the page load.
After recording, select an area of interest in the overview by dragging.
Then, zoom and pan the timeline with the mousewheel or WASD keys.
Learn more


Elements
Console
Sources
Memory
Profiles
Select profiling type
Heap snapshot
Heap snapshot profiles show memory distribution among your page's JavaScript objects and related DOM nodes.
Allocation instrumentation on timeline
Allocation timelines show instrumented JavaScript memory allocations over time. Once profile is recorded you can select a time interval to see objects that were allocated within it and still alive by the end of recording. Use this profile type to isolate memory leaks.
Record allocation stacks (extra performance overhead)
Allocation sampling
Record memory allocations using sampling method. This profile type has minimal performance overhead and can be used for long running operations. It provides good approximation of allocations broken down by JavaScript execution stack.
Select JavaScript VM instance
3.5 MB / 5.3 MB (empty)


Elements
Console
Sources
Network
Security
Overview
Main origin
Reload to view details
Security overview
This page is secure (valid HTTPS).
Certificate - valid and trusted
The connection to this site is using a valid, trusted server certificate issued by DigiCert SHA2 Secure Server CA.
View certificate
Connection - secure connection settings
The connection to this site is encrypted and authenticated using TLS 1.2, ECDHE_RSA with P-256, and AES_128_GCM.
Resources - all served securely
All resources on this page are served securely.


Elements
Console
Sources
Security
Overview
Main origin
Reload to view details
Security overview
This page is not secure.


Elements
Console
Sources
Network
Audits
(new audit)
Audits
Identify and fix common problems that affect your site's performance, accessibility, and user experience. Learn more
Device
Mobile
Desktop
Audits
Performance
Progressive Web App
Best practices
Accessibility
SEO
Throttling
Simulated Fast 3G, 4x CPU Slowdown
Applied Fast 3G, 4x CPU Slowdown
No throttling
Clear storage
Run audits



CHAPTER 10

Debugging JavaScript Applications

"If debugging is the process of removing software bugs, then programming must be the process of putting them in."

~ Edsger Dijkstra, a Dutch programmer, systems scientist.

Debugging is a very important aspect of programming, equally important to coding. Debugging enables you to detect the real cause of the bug, the part of code and the code flow that is causing that bug and what kind of data is causing the issue. In order to become a competent programmer in any technology, we need to understand the process of debugging otherwise we will be guessing the solution and spending a lot of time in hit-and-trial. It is next to impossible to write a perfectly working code, the first time through with zero bugs, even for an expert

programmer. So, irrespective of the fact that you are a novice or a subject matter expert, your efficiency in writing good code will be directly impacted by your efficiency in debugging your code the right way.

With the evolution of web application development, the debugging aspect has also evolved to cater to the requirements of this important process.

Structure

- The browser devtools
- How to launch the browser devtools?
- Parts of the Chrome devtools
- The debugging processes

Objective

At the end of this chapter, you will get an understanding of the debugging tools of web development and how you can use it to debug your application.

The browser devtools

The browser devtools are the fundamental tools for a front-end developer's toolset which provide many features to make debugging structured and methodical. They are provided in all the modern browsers to give a peek into how the information is being rendered on the page, how the related values are getting changed, how the data is getting fetched from an external source, how performant and secure your application is, and so on.

How to launch the browser devtools?

The browser devtools open up as a child window of the main window as shown in the following screenshot:

Figure 10.1: *The browser devtools opens as a sub-window*

The devtools can be launched using the following options:

- **Keyboard shortcuts:**
- **Internet Explorer and Edge**: *F12*
- **macOS:** ⌘ + ⌥ + I
- **All other browsers:** *Ctrl* + *Shift* + *I*
- **Menu options:**
- Firefox: **`Menu | Web Developer | Toggle Tools`**, or **`Tools | Web Developer | Toggle Tools`**.

- Chrome: **`More tools | Developer tools.`**
- Safari: **`Develop | Show Web Inspector`**. If you can't see the **`Develop`** menu, go to **`Safari | Preferences | Advanced`**, and check the **`Show Develop`** menu in the menu bar checkbox.
- Opera: **`Developer | Developer tools.`**
- Edge: **`Menu | Developer tools.`**
- **Context menu:** Press and hold or right click on an item on a webpage (*Ctrl*-click on the Mac) and choose **`Inspect Element`** from the context menu that appears. An additional advantage with this approach is that it directly highlights the code of the element on which you clicked to inspect. You can refer to the following screenshot:

Figure 10.2: *Using Context Menu to Inspect element*

In this book, we will explore the Chrome devtools, as Chrome is the most popular browser which has a rich feature set, good performance and also easy to use. Most of these features do exist for the other browser devtools also, and you can explore them if you work on a different browser.

Once you open the browser devtools using any of the given options, we can set the position of the devtools by using the **`Dock side`** option which you get when you click on the three vertical dots icon as shown in the following screenshot on the top right-hand corner of the page:

Figure 10.3: *Chrome devtoolsposition can be set using Dock side*

Let's now explore each of the tabs shown in *Figure 10.3* for the Chrome devtools and see what capabilities they provide.

Parts of the Chrome devtools

Each of the tabs of the Chrome devtools handles a different aspect of debugging and provides details regarding that specific aspect.

Elements and styles

This section gives a complete view of the HTML elements of the application with the complete DOM hierarchy. You can investigate and see all the HTML elements appearing on the page.

The styles sub-tab shows the CSS styles applied to the selected element. In the styles tab, it shows the different levels of cascading styles which are applied to the element. It also has the Box-model view which shows the detailed view of the selected element with its margin, border, padding, and the actual content.

Another powerful capability provided here is Live Edit. We can edit the

HTML as well as apply styles on the go as we are inspecting the element to see how it changes the live view. As a part of this, you can change elements, drag and change their order, delete the elements from the view, add new style properties or change existing values applied to styles at each element level. The live view reflects whatever changes you make but all these changes are lost if you reload your page as this is not saved in the source code.

The following screenshot shows the **`Elements`** tab along with the styling details and the border box model:

Figure 10.4: *Elements and Styles in Chrome devtools*

Console

In the **`Console`** tab, we can see the error messages which are encountered in the application. Additionally, we can add logs to our JavaScript code to view the values of different variables in our code using `console.log` statements.

On the right of each of the console messages, the source code (file) name of the message is provided as a link. When you click on the link, you are taken to the source code in the **`Sources`** tab which is the origin of the message.

The **`Console`** is also like a command line environment which has the REPL capability, which means it can be used for Read, Evaluate, Print, and Loop. You can execute any JavaScript statements in the **`Console`** and view the results as shown in the following screenshot:

Figure 10.5: *Console as REPL in Chrome devtools*

Sources

This tab will give access to your application code to which you can add breakpoints and enable the JavaScript debugger. You can view the complete project folder structure here in the top part and have access to the application source code files. The bottom section is the JavaScript debugger which enables you to step into the code and view the data value changes. Refer to the following screenshot, which shows the **`Sources`** tab:

Figure 10.6: *The Sources tab*

Network

This tab will give a sneak peek into the network calls and data fetch which are made in the application. We can investigate this further to check whether the call was successful and whether the proper response was

received. Refer to the following screenshot showing the **Network** tab:

Figure 10.7: *The Network tab*

Application

This tab gives the details of storage, especially the local storage and session storage which are commonly used to handle data in the application locally or for a session, which is shown in the following screenshot:

Figure 10.8: *The Application tab*

Performance

The **Performance** tab provides the capability to do performance profiling of the application. The profiling can be done in the Incognito mode of the browser to ensure clean browser and adjusting the options using settings. Using the record option, the performance statistics can be gathered and analyzed by starting and stopping the recording while navigating the application. The data collected will show the time spent in various activities and analyze which will give insights into the application if it is spending more time in some bottleneck activities. Once the bottlenecks are identified, steps can be taken to remove bottlenecks and improve the performance. Refer to the following screenshot that shows the options of the **Performance** tab:

Figure 10.9: *The Performance tab*

Memory

The **Memory** tab enables you to find how memory is being allocated in your application and get an insight into any memory issues that may affect page performance due to memory leaks, bloats, and frequent garbage collections shown as follows:

Figure 10.10: *The Memory tab*

Security

The **Security** panel is the tab in devtools which helps in inspecting the security of a page; whether it is protected by HTTPS or not, as shown in the following screenshot:

Figure 10.11

If not secure with HTTPS, it gives an appropriate indication as follows:

Figure 10.12: *The Security tab*

Audit

A new Audits panel (**https://developers.google.com/web/updates/2017/05/devtools-release-notes#lighthouse**) was included in the Chrome version 60 which audits your application and provides scores for progressive web apps, performance, accessibility, and best practices. This is enabled by an open source called **Lighthouse** which gives a report which can be used to improve the quality of your application. The following screenshot shows the options available in the **`Audit`** tab of Chrome devtools:

Figure 10.13: *The Audit tab*

The debugging process

How the devtools are put to use and to investigate the problem in the code effectively is what defines the debugging process. The issue is replicated by enabling debugging steps, and then the code is looked into to locate the part of code, introducing the issue.

Using console.log statements

When we start programming with JavaScript and we are introduced to the console.log statement, it seems like it is the most powerful tool to be able to view the values at runtime and hence, becomes the default weapon for debugging.

Just logging out the values gives us the visibility to see how the values have passed and changed through the execution of the application logic wherever we add log messages. While this gets us started, we do need to be careful and remove all the logs before the production deployment. Also, this is not an efficient way of debugging and it is limited to only as many log messages we add.

Using the JavaScript Debugger

The JavaScript debugger is the most effective and efficient debugging tool which gives complete insight into the source code. By setting breakpoints into the code, you can pause the code in the middle of the execution and view the variable values in scope at that point in time and traverse through the application code line by line as the execution proceeds. In order to debug your problem, you will have to replicate the error scenario, set up appropriate breakpoints to investigate the values, and find the issue.

How to add breakpoints?

Go to the **`Source`** tab and navigate to your project folder. You can view

all your code files. The top-left section is the project navigator, the top-right section is the project view editor, and the bottom section is the JavaScript debugger section.

Breakpoints can be added in the JavaScript debugger to break on specific lines of code or specific actions like Data fetch, DOM changes, exceptions, or events. The following are the different types of breakpoints and we can see how each of them can be included in the debugging process:

- **`XHR/fetch Breakpoints`** can be added on **`XmlHttpRequest`** in order to debug AJAX requests. To add this breakpoint, select the XHR Breakpoints option, click on the plus (+) icon, and enter a part or all the fetch URL on which you want to introduce a breakpoint. Alternatively, you can select the Any XHR or fetch option to break at every fetch. The following screenshot shows the option where XHR breakpoints can be added:

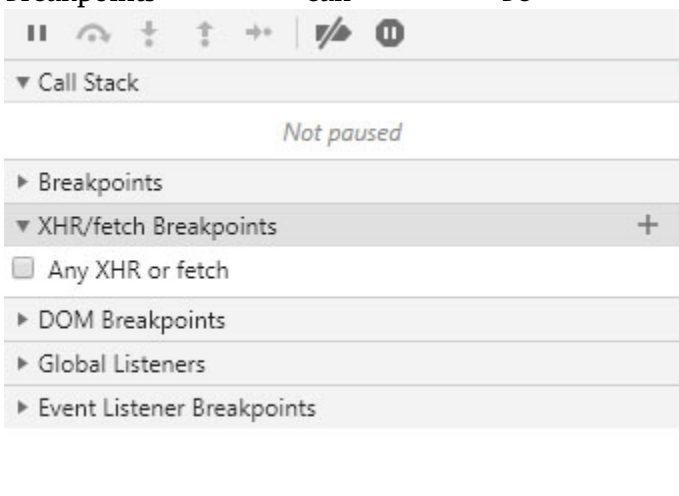


Figure 10.14: *Adding XHR/fetch breakpoints*

- **`DOM Breakpoints`** can be added by going to the **`Elements`** tab, right clicking on the **`DOM element`** and selecting the nature of breakpoint. The breakpoint can be on DOM **`subtree modifications`**, element **`attribute modifications`** or **`node removal`** from DOM, as shown in the following screenshot:

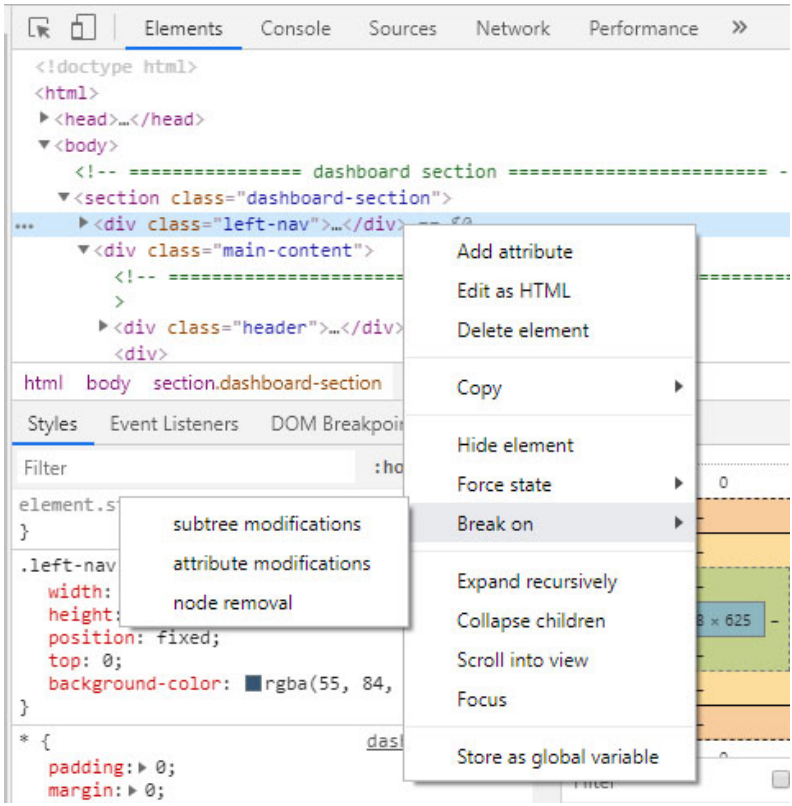


Figure 10.15: *Adding the DOM breakpoint*

- Another special breakpoint is **`Event Listener Breakpoints`** which can be added on a specific event like mouse click, keyboard button press by selecting the specific action from the list provided in the debugger section of the sources tab, as shown in the following

screenshot:

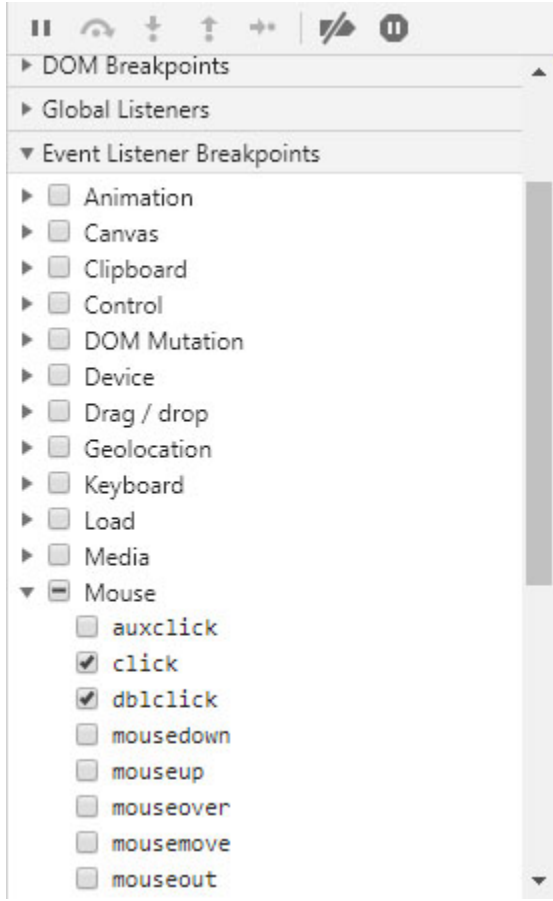


Figure 10.16: *Adding Event Listener Breakpoint*

- **Line of code breakpoints:** In order to add a general breakpoint in the flow of your source code, you can click on the left-hand side of the line of code in the source code view panel to which you want to add the breakpoint. It gets highlighted as a solid blue arrow, and the line gets listed in the **`Breakpoints`** section of the debugger as shown in the following screenshot:

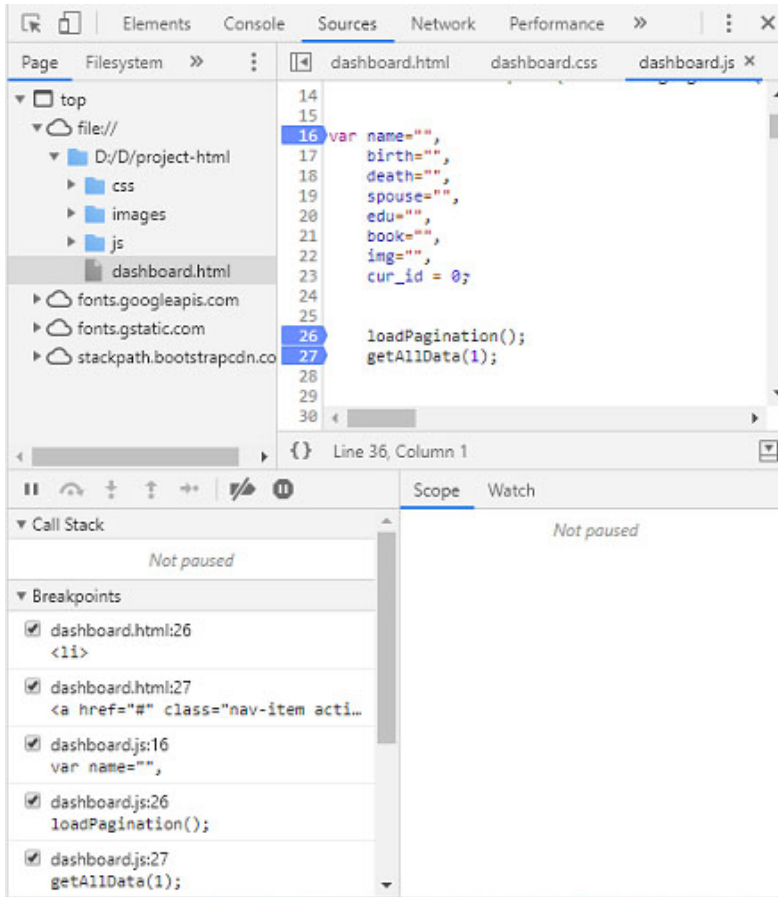


Figure 10.17: *Adding Line of Code Breakpoints*

Once the breakpoints are added, you can reload your application. The application execution will stop at the breakpoint and you will see the following message with two icons to control the debugger:

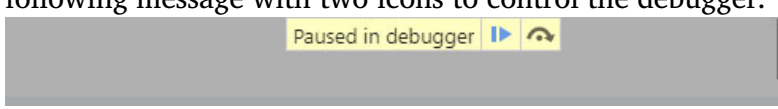


Figure 10.18: *The application paused by the debugger*

The other control options (the first two are the same as the preceding screenshot) are also available on the **`Sources`** tab in the debugger section as shown in the following screenshot:

Figure 10.19: *Debugger control options*

Let's understand each of the Figure 10.19 icons:

- **Resume script execution** **:** The first control will continue with

the application execution and take you to the next breakpoint if any. If there are no other breakpoints, it stops the debugger and the application execution continues.

- **Step over next function call** **:** This control will take you to the next function call. It will take you to each function call but not line-by-line into the function execution. This can be used when we know that the problem is not inside the function and we just want to proceed with the function execution.

- **Step into next function call** **:** This control will take you through the execution of the function, one line at a time. This can be used when we know the bug lies inside the function and we want to investigate it line by line. From the Chrome 65 version, stepping into asynchronous code will pause for the asynchronous code to run and return to continue execution. For older versions, step into option, would step over the async function and move onto the next line.

- **Step out of current function** **:** This control can be used to continue with the execution of the function which you had stepped into to investigate in detail. Once you are done with the investigation, you can step out to execute the remaining function code.

- **Step** **:** This is added in the newer version (65) and it behaves like the old version step into, that s, it will step over the async function and move onto the next line.

The last two controls are used to:

- Deactivate all the breakpoints and continue with normal execution
- Pause on exceptions

Once the application is paused at a breakpoint and you can investigate into the values of the source code by hovering onto the variables in the current scope as shown in *Figure 10.20*:

Figure 10.20: *Debugging in process (investigating values)*

For further investigation into the values, you can use the following sections of the debugger:

- **Scope pane:** This will show all the active local and global variables at the paused line of code with their values.
- **Watch pane:** This can be used to add any variable or expression and monitor how its value changes with each step. This helps in detailed investigation to find the exact step where the wrong value is getting introduced.

- **Console tab:** You can also make use of the REPL property of the console pane to evaluate and print values based on the variables active at the breakpoint. This is the ideal place to try out the code fix and see if it is giving the expected result.

Once you are done with the debugging, you can use the resume icon until there are no more breakpoints and the debugger stops. By setting proper breakpoints and reproducing the issue, you should be able to look into the values of your code and locate the problem.

Using the various capabilities provided by the devtools, you can have complete control of the debugging process.

Conclusion

Irrespective of whether you are building a vanilla JavaScript application or using any of the modern JS Frameworks, the debugging process is made effective with the use of the browser devtools, especially the Debugger. With efficient debugging, we are better equipped to approach the problem at hand and find a suitable solution. Having learned about the debugging process, in the next chapter, we will learn about another important aspect of the development lifecycle-Unit Testing and its automation.

Questions

1. Which of these statements regarding breakpoints is correct?

   A. DOM breakpoints can be added from the Sources tab.

   B. Event Listener Breakpoint can be added from the Elements tab.

   C. You can select the Any XHR or fetch option under the Sources tab to break at every fetch.

   D. None of the above.

   Answer: Option C.

   DOM Breakpoints are added from the Elements tab, Event Listener Breakpoints are added from the Debugger section of the sources tab. XHR Fetch breakpoints are added from the Sources tab using the plus button. You can select the Any XHR or fetch option under the Sources tab to break at every fetch.

2. The data collected will show the time spent in various activities and analyze which will give insights into the application if it is spending more time in some bottleneck activities. Which tab facilitates this activity?

   A. Sources tab

B. Performance tab

C. Memory tab

D. Audit tab

Answer: Option B.

The data collected will show the time spent in various activities and analyze which will give insights into the application if it is spending more time in some bottleneck activities. Once bottlenecks are identified by looking at this data in the performance tab, steps can be taken to remove bottlenecks and improve performance of the application.

3. This can be used to add any variable or expression and monitor how its value changes with each step. Which pane are we talking about?

A. Console pane

B. Scope pane

C. Memory pane

D. Watch pane

Answer: Option D.

Watch pane can be used to add any variable or expression and monitor how its value changes with each step. This helps in detailed investigation to find the exact step where the wrong value is getting introduced.

4. How can you launch the devtools for debugging your application?

A. Keyboard shortcuts.

B. Menu options

C. Context menu

D. All of the above

Answer: Option D.

The devtoolscan be launched using keyboard shortcuts, menu options as well as using the context menu on right clicking on a specific element.

5. Which of these is not a valid tab of browser devtools?

A. Audit

B. Performance

C. Context

D. Network

Answer: Option C.

Context is not a valid tab name in browser devtools. The valid tabs include Console, Sources, Elements, Network, Performance, Memory, Security, and Audit.

```
it('Existing ToDo task can be removed', ()=>{
      let todolist = new ToDo();
      let task1= {
        id: 1,
        title: "Read a book",
        complete: false
       }
       let task2 = {
        id: 2,
        title: "Finish the chapter",
        complete: false
       }
       todolist.addTodo(task1)
       todolist.addTodo(task2)//Now the list has 2 tasks
      todo.delete(1) // Delete the 1st item
      expect(todo.getItems().length).toBe(1);
    })
```

CHAPTER 11

Unit Testing Automation

"Why do we never have time to do it right, but always have time to do it over?"

~ Anonymous

In the last few chapters, we learned about JavaScript and how to build and debug applications in JavaScript. Another important aspect of any kind of software development is testing. Testing is the process of checking whether our application is working the way it is intended to do. A well planned and executed testing phase ensures a good product delivered to the customer.

In this chapter, we will look at one of the aspects of testing - Unit Testing. We will understand what unit testing is, why unit testing should form an

integral part of development, and how it can be automated using unit testing automation tools available for JS applications.

Structure

- Unit testing
- Introduction to Jasmine
- Jasmine basic constructs for synchronous testing
- Asynchronous testing

Objective

At the end of this chapter, you will learn all about unit testing and how it can be automated for JavaScript applications.

Unit testing

According to ANSI/IEEE 1059 standard, Software Testing is defined as *"A process of analyzing a software item to detect the differences between existing and required conditions (that is, defects) and to evaluate the features of the software item."*

Unit testing is testing each module or component of the application individually to check whether they are working as expected. Unit Testing is the first testing performed by the developer and it is an integral part of development and it should be planned and executed in conjunction with development. This makes it essential to introduce unit testing to you at this point when you are starting with learning and development.

The first step of unit testing is identifying all the business scenarios and use cases for the component. Unit testing can be done manually by executing each business scenario for the component and comparing the actual results with the expected results. As a part of manual unit testing, you should list out all the different test cases to cover the different user stories around your component. For each test case, identify the inputs and expected outputs. Then, manually execute each test case and make a note of the actual results. Comparing the actual with expected will tell you whether the test case is successful (results equal) or a failure (results don't match).

The results help you, as a developer, to identify the issues and bugs in your code before it goes through any formal testing. Being a developer, you should understand the importance of effective unit testing which results in delivering good quality end product. The sooner a bug is identified, the cheaper it proves to be. You should aim at delivering a completely bug-free component from your end.

The process of testing is structured as it expects some set of inputs and gives some set of outputs. If the given output matches with the expected output, then the test scenario is successful else it has failed. By following the same structure and approach, it is possible to automate the testing process by defining the different test scenarios with inputs and expected outputs.

The process of unit testing can be automated with the use of JavaScript frameworks like Jasmine. Automating the unit testing embeds the process into the development lifecycle and also saves a lot of effort in order to retest the same scenarios after any change.

In this chapter, you will be introduced to Jasmine, one of the most popular testing frameworks, which forms a good foundation to automation unit testing.

Introduction to Jasmine

Jasmine is a self-sufficient open source framework which operates without any external dependencies and is used for testing JavaScript code. It is behavior-driven and has a very easy to understand constructs which make unit test automation very convenient. It does not require the DOM to perform testing and automation related to the DOM.

Let's get started by setting up Jasmine.

Jasmine setup and configuration

Now we will see how we can setup Jasmine for our testing automation needs:

1. Install Jasmine globally so it is available for all your JS applications as shown in the following command:

   ```
   npm install –g jasmine
   ```

   The following screenshot shows the command prompt screen after running the above command in windows system:

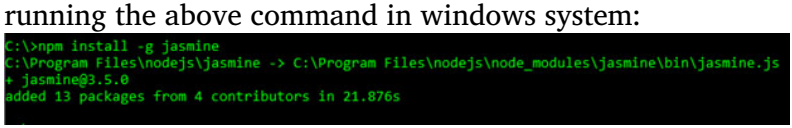
   
   C:\>npm install -g jasmine
C:\Program Files\nodejs\jasmine -> C:\Program Files\nodejs\node_modules\jasmine\bin\jasmine.js
+ jasmine@3.5.0
added 13 packages from 4 contributors in 21.876s

   Figure 11.1

2. Once installed, you will need to setup the configuration in a JSON file to list out the source files, helpers and spec files to be executed. To setup Jasmine for your existing project, you can use the following command:

   ```
   jasmine init
   ```

The following screenshot shows the command prompt screen after running the above command in windows system:

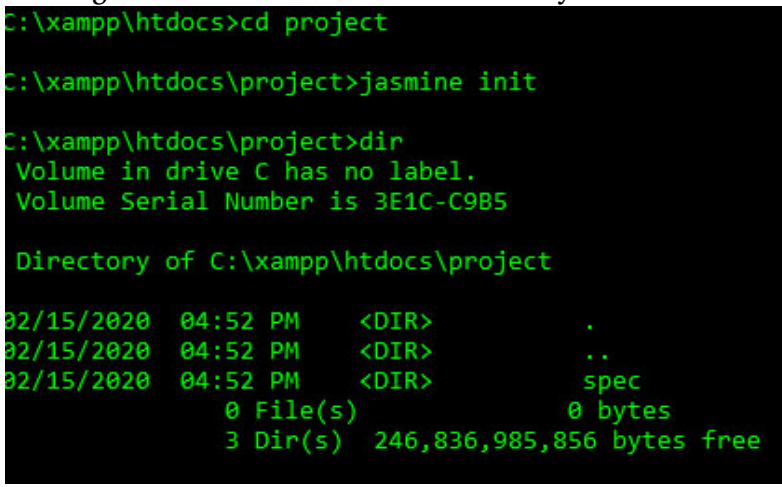


Figure 11.2

This command will create a spec folder as shown and a `jasmine.json` configuration file inside it with basic configuration for Jasmine.

The `jasmine.json` file looks like the following code:

```
{
 // Test Specification directory path relative to
 the current path.
 "spec_dir": "spec",
 // Array of file paths relative to spec_dir to
 include and exclude(Paths starting with ! are
 excluded)
 "spec_files": [
  "**/*[sS]pec.js",
  "!**/*nospec.js"
 ],
 // Array of helper file paths relative to spec_dir
 which will be included before the spec files
 "helpers": [
  "helpers/**/*.js"
 ],
 // Stop execution of a spec after the first
 expectation failure in it
 "stopSpecOnExpectationFailure": false,
 // Run specs in semi-random order
 "random": false
}
```

3. After setting up your `jasmine.json`, you are ready to create and execute test spec files, just by running Jasmine from the root folder.

```
jasmine MySpecFile.spec
```

There are other options to change the configuration options from the command line for Jasmine. With the basic configuration setup, we can start with writing the Jasmine test specification file.

Next we will look at the constructs which are used to write the test specifications.

Jasmine basic constructs

In order to start writing the test specs, you need to first understand the basic constructs of Jasmine which are used to write the test cases.

We can build our automation test scripts by using the following Jasmine constructs.

describe

This construct explains what we are trying to do in the testing. We will start defining the test specifications by grouping them in test suites. All the related test cases are grouped under a test suite which is defined using the describe function as shown in the following code:

```
//This is my first test suite
describe("My Test Suite for first component",
function() {
 //…..Everything comes inside this
});
```

In the preceding code, the describe function takes two parameters: the first one represents the title of the test suite and second parameter is the function that contains the implementation details of the test suite.

it

The test suite implementation will include one or more test cases which are related to the test suite.

The function it is used to give the implementation of each test case, which takes two parameters as the title of the test case and the second parameter includes the functional implementation of the test case.

The following test suite shows the usage of it function:

```
//This is test suite
describe("My Test Suite for first component",
```

```
function() {
  it("My first test case for first component",
  function() {
   expect( expression ).toEqual(true);
  });
 it("My second test case for first component",
 function() {
   expect( expression ).toEqual(true);
  });
});
```

We should make use of it to give clear name and implementation of each test case.

beforeEach and afterEach

Any logic to be executed to set up data for the test suite before the execution of each test case can be done in the `beforeEach` function. Setting up environment-related variables is done in this function.

Any cleanup or closure-related activities can be performed in the `afterEach` function, which is called once after each test case.

beforeAll and afterAll

Any common setup and clear up logic can be put in the functions beforeAll and afterAll, which are executed only once for all test specs before and after, respectively.

expect

The `expect()` function is used to create an expectation that takes the actual values, which can be the result of another expression expected from the test case. This is the main part of the test specification and is used in conjunction with the matchers to design the test cases. It states what is the result expected from the following piece of code:

```
expect( expression ).toEqual(true);
```

This construct is where we will do the comparison of values to get a result.

Matchers

Matchers are used to compare two values: the expected value and the actual value. It then returns a `true` or `false`, which becomes the result of the test case.

In the earlier example, we saw the use of the `toEqual` function to compare two values. Similarly, Jasmine has a variety of matcher functions

which help in writing test cases to match different values and decide whether the test case is a pass or not.

The following table provides a list of the matchers which can be used to validate results in testing using Jasmine:

Matcher	Usage
`toBe()`	Works like `===`. Compares value and type to be equal
`toEqual()`	Checks the equality of simple strings, objects and variables
`toMatch()`	Finds if a value matches a string or a regular expression pattern
`toBeDefined()`	To check whether a value is defined
`toBeUndefined()`	To check whether a value is undefined
`toBeNull()`	To check whether a value is null
`toBeTruthy()`	To check whether a value is `true`
`toBeFalsy()`	To check whether a value is `false`
`toContain()`	To check whether a substring exists in a string or an item exists in an array
`toBeLessThan()`	Mathematical less than comparison
`toBeGreaterThan()`	Mathematical greater than comparison
`toBeCloseTo()`	Mathematical precision comparison
`toThrow()`	To check whether the function throws an error
`toThrowError()`	To check whether a specific error is thrown
`not`	Can be used in conjunction with any of the above to negate the effect of the condition

Table 11.1: *Matchers of Jasmine*

Using these different matchers, you can construct your condition to be checked in the test cases.

Let's see with help of an example on writing test cases for the Todo App.

The test suite will check the following test cases:

- If a new ToDo task can be added
- If the existing ToDo task can be removed from the list

The test suite structure will look like the following code:

```
describe('ToDo function test suite', ()=>{
    it('New ToDo task can be added', ()=>{
       //…
    })
    it('Existing ToDo task can be removed ', ()=>{
       //…
    })
    })
```

Let's see how each test case will be written:

```
it('New ToDo task can be added', ()=>{
      let todolist = new ToDo();
      let task= {
       id:1,
```

```
    title: "Read a book",
    complete: false
  }
    todolist.addTodo(item);
    expect(todolist.getItems().length).toBe(1);
  })
```

We manually added one task to the `todolist`, so the length of the list should be 1. Also, it establishes the test case that we are able to add tasks using the ToDo functionality.

Similarly, to check the remove functionality, we need to have our `todolist` with a predefined hardcoded list of tasks. Then, using the `remove()` function, we will remove items. Lastly, we can compare the list length to the expected size:

We manually perform the steps to use the functionality to be tested and then compare the expected result with the actual result. Once done, the automated test scripts can be rerun as many times you want to unit test your component/functionality.

Many times there are scenarios where the function/component to be tested is dependent on some other components. In unit testing, we would not want to be dependent on other components but be able to test our component standalone. The dependencies could be on external functions, network connections, database, files, and so on, so there should be a generic way to replace and mock any dependency.

The next set of constructs provides this mocking capability to facilitate unit testing without dependencies.

Spies are the test double functions which can stub any function calls and handle it as required. It can check whether the method was called and handle the call to return mocked values or internally invoke mocked functions.

The following table lists out the matchers and functions which can be used to spy and provide different behaviors by handling dependencies appropriately:

Method	Usage
`spyOn()`	This spies on the objMethod, which should be a method of the myObject instance as shown: `spyOn(myObject, 'objMethod'`
`toHaveBeenCalled()`	This matcher is used to verify if the method was called: `expect(myObject.objMethod).toHaveBeenCalled()`
`toHaveBeenCalledWith()`	This matcher is used to check whether the method was called with specific arguments: `expect(myObject.objmethod).toHaveBeenCalledWith('arg1','arg2')`
`and.callThrough()`	This function is chained with the spy to continue the call to the original function, in spite of the spy: `spyOn(myObject, 'objMethod').and.callThrough();`
`and.returnValue()`	This function is chained with a spy to return a specific value whenever the function is

invoked:

```
spyOn(myObject, 'objMethod').and.returnValue('Return Value');
```

This function is chained with a spy to invoke another function passed as the parameter here, instead of the original function:

```
spyOn(myObject, 'objMethod').and.callFake(function(arg1, arg2) {
    return 'this is response from fake function' ;
 });
```

This function is chained with a spy to throw an error every time the spy is called:

```
spyOn(myObject, 'objMethod').and.throwError('Errored out!');
```

This function chained with a spy will return true if the method has been called at least once, else will return false.

This function will return the number of times the method has been called.

This function returns a reference and arguments to the most recent method invocation.

This function will reset and clear whatever data is being tracked by the spy on the method.

This function returns the arguments for the nth call to the method.

This function returns the arguments to all the calls made to the method.

This function returns a reference and arguments to the first call made to the method.

This function returns all the reference and arguments of all the calls made to the method.

This function can be used to create a spy when a real function does not exist:

```
dummy= jasmine.createSpy('dummy');
It will have access to all spy-related functionalities without
having a real function implementation behind it. You can chain it
with other methods to mock return values/functionalities as
follows:
dummy.and.returnValue('Dummy Return Value');
```

This function can be used to create a complete object with multiple spy methods for different functionalities of the object without any implementations:

```
dog = jasmine.createSpyObj('dog', ['bark', 'play', 'eat','
sleep']);
 dog.bark();
 dog.eat();
 //…each of these are spy methods on the dog object
```

Table 11.2: *Spy-related functionality*

Using the spy functionality, methods can be tracked and mocked to return different return values or call different functions. This can be used to mock any dependencies in our functionality and be able to continue with independent unit testing.

In the next topic, you will learn another aspect of unit testing automation, that is, asynchronous testing.

Asynchronous testing

In *Chapter 7, Asynchronous JS*, you learned about synchronous and asynchronous programming and the different aspects of asynchronous JavaScript. Testing synchronous code is more straightforward as it will be executed step by step. Whatever constructs we have seen so far can be used for testing synchronous code, but testing of asynchronous constructs

like callbacks, promises, `async-await` will require you to replicate the asynchronous behavior by inducing some extra weight time for the process during the automation test cases.

The Jasmine spy constructs along with asynchronous JavaScript can be used to handle and automate asynchronous behavior.

Let's see some of the scenarios here:

- To mock and handle an asynchronous function which returns a promise:

  It is important to call the done() function inside the asynchronous test case to inform the test runner that it's the end of the current test case and it can proceed with the next one.

  The below code tests myTestService which returns a promise as response, which is handled and compared with expected result as shown:

```
describe('getPromiseData function', function () {
  const arg1= 'dummy';
 it('should call getPromiseData from
 myTestService', function(done) {
    spyOn(myTestService,
    'getPromiseData').and.returnValue(Promise.resolve(
//promiseData is some json data which is being
returned as a promise
   myTestService.getPromiseData(arg1)
    .then((result) => {
     expect(myTestService.getPromiseData).toHaveBeenCa
     expect(result).toEqual(promiseData);
     done();
 });
 });
});
```

  Dummy data is returned as a promise result for `getPromiseData` function of `myTestService` using `spyOn`. The result is handled using then block and compared with expected result to mock the test case.

- **To test a real asynchronous function call using async-await:**

  The `async-await` functionality can be used whenever there is an asynchronous call to wait and get the result verified as shown in the following code:

```
it('test myAsyncFunction', async function() {
 const result = await myAsyncFunction();
 expect(result).toEqual(someExpectedValue);
```

```
});
```

The following test case is to execute an asynchronous function named `myAsyncFunction` and compare the result with some expected value.

Similarly, the other spy methods can be used in combination with promises and `async-await` to mock asynchronous behavior.

- **Changing asynchronous behavior by inducing wait time using mock clock and making it synchronous:**

  Using the Jasmine mock clock, the wait time for any asynchronous function can be mocked so that it runs in synchronous mode and can be tested easily without any extra wait time.

  The below code shows how we can mock wait time using `jasmine.clock().tick` and make asynchronous code to run in synchronous mode:

```
//This is the inner function which takes a callback
which is called after 10ms
 function myAsyncFunction(innerFunc) {
  setTimeout(function() {
   innerFunc('dummy');
 }, 1000);
}
describe('myAsyncFunction', function() {
 beforeEach(function() {
//this will setup the mock clock
  jasmine.clock().install();
 });

 afterEach(function() {
//this will stop or uninstall the mock clock
  jasmine.clock().uninstall();
 });

 it('myAsyncFunction takes 10 ms', function() {
  const innerFunc= jasmine.createSpy('innerFunc');
  myAsyncFunction(innerFunc);
  jasmine.clock().tick(1000);//mock the wait for
  10ms to make it synchronous
  expect(callback).toHaveBeenCalledWith('dummy');
 });
});
```

In order to use the `jasmine.clock()`, it needs to be installed in `beforeEach` function and uninstalled in `afterEach` function after use. It can be used to mock the 1000s wait time in the preceding code, which is

the time taken by the asynchronous function `myAsyncFunction`, so that it feels like a synchronous process overall.

Using the preceding listed methods and approaches, any scenario can be automated and tested to verify the functionality.

Conclusion

In this chapter, we learned the different aspects of unit testing automation of JavaScript applications using Jasmine. Unit testing is a very important aspect of development which ensures the code is behaving in the expected way. As a part of unit testing, all the expected test scenarios are scripted and tested to ensure basic behavior compliance. Jasmine can be used to automate both synchronous and asynchronous JavaScript code, improve the efficiency of the development process and also the code effectiveness and quality.

Now that we have learned to develop, debug, and test your JavaScript applications, in the next chapter, we will learn how to build and deploy your JavaScript applications.

Questions

1. Which matcher is used to verify if the method was called?

    A. toHaveBeenCalled()

    B. and.callThrough()

    C. and.callFake()

    D. None of the above

    Answer: Option A.

2. What aspect of Jasmine can be used to convert asynchronous behavior to synchronous?

    A. jasmine.await()

    B. jasmine.spyOn()

    C. jasmine.clock()

    D. Not possible to be done

    Answer:Option C.

3. All the related test cases are grouped under a test suite which is defined using the ____ function.

    A. desc

    B. it

    C. describe

D. group

Answer: Option C.

4. The spy methods cannot be used with asynchronous functions.

   A. TRUE

   B. FALSE

   Answer: Option B.

5. This function is chained with the spy to continue the call to the original function.

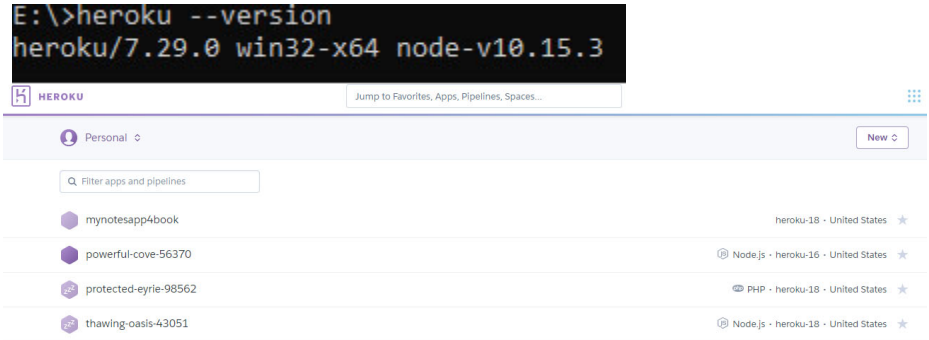
   A. and.callThrough

   B. and.callFake

   C. and.callsAny

   D. and.calls

   Answer: Option A.



Chapter 12

Build and Deploy an Application

"We know only too well that what we are doing is nothing more than a drop in the ocean. But if the drop were not there, the ocean would be missing something."

— Mother Teresa

Deployment is the final step to take our application to the world. In the previous chapter, we built a simple JavaScript application for handling user entered notes. In this chapter, you will learn how to deploy your application and make it available on the internet so that anyone can access it using a URL. There are many platforms available for this, but we will be using Heroku for the deployments explained in this book. Heroku is a cloud platform which provides the fastest way to make an application available on a URL and accessible on the internet.

In this chapter, you will learn how to deploy your applications to the cloud using Heroku so that they are accessible over the internet using a URL.

Structure

- Why deployment?
- Getting ready for deployment
- Detailed steps for hosting

Objective

At the end of this chapter, you will get an understanding of the process of

deployment to make the application available on the internet.

Why deployment?

In the last few chapters, you learned about different aspects of developing an application. When you are developing on your local system, your application is only available to you. In a real project, will you really develop for yourself, or you would want your application to reach out to as many people as possible. You can make your application accessible over the internet by deploying it on the cloud. This process of deployment is provided by many platforms; Heroku being one of the most popular ones, as it makes the process very easy. Let's learn how you can deploy your application to the Heroku cloud.

Getting ready for deployment

You have built an application using JavaScript or React or any other framework. After having tested it thoroughly for the requirements, you would want to deploy it to the cloud and showcase your work to the entire world. Deployment is all about that. With cloud providing all the infrastructure, it has become very easy and freely available too.

You have an application which is working and ready to be deployed. In order to proceed with deployment, you will first have to make the prerequisites ready.

Let's first make sure we have the prerequisites for deployment:

1. In this entire book, you have been using `npm` for the installations, and you will do the same here. So, you already have the first prerequisite - Node and NPM.
2. You can check the versions using the following commands:
    - `node -v`
    - `npm -v`

5. Git will be used internally by Heroku to maintain the versions of the deployed code; hence, you will need Git. Install it from **https://git-scm.com/downloads**.
6. You will be using Heroku, so first you will need to sign up for free hosting at **https://signup.heroku.com/**.
7. You get the following form as you visit the Heroku website for sign

up:

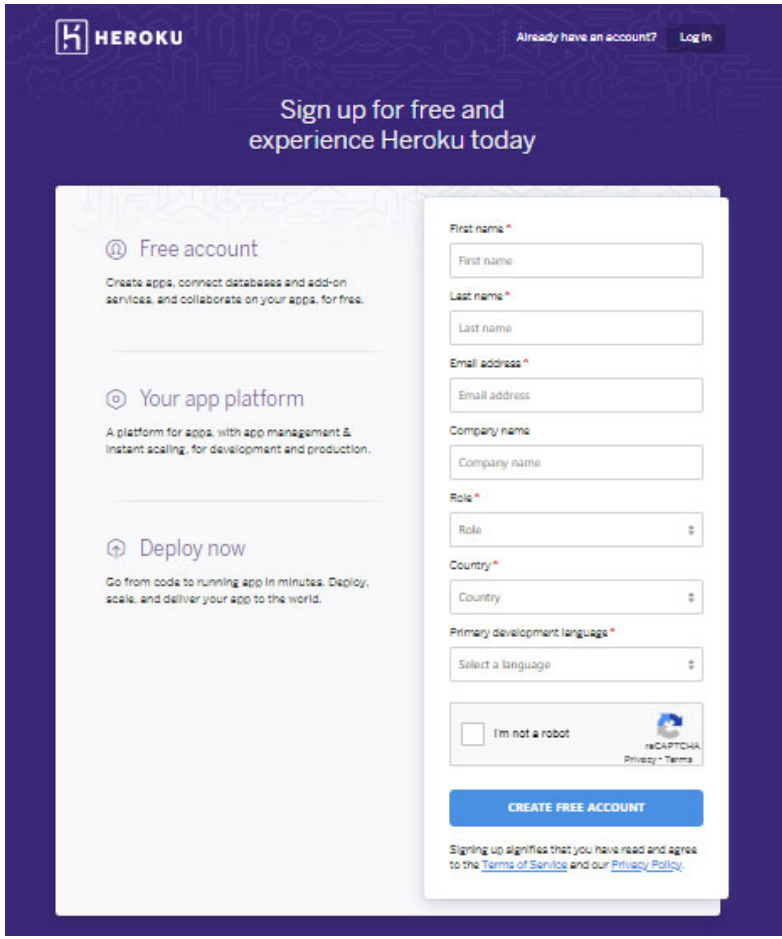

HEROKU
Already have an account?
Log In
Sign up for free and
experience Heroku today
Free account
Create apps, connect databases and add-on services, and collaborate on your apps, for free.
Your app platform
A platform for apps, with app management & instant scaling, for development and production.
Deploy now
Go from code to running app in minutes. Deploy, scale, and deliver your app to the world.
First name *
First name
Last name *
Last name
Email address *
Email address
Company name
Company name
Role *
Role
Country *
Country
Primary development language *
Select a language
I'm not a robot
reCAPTCHA
Privacy - Terms
CREATE FREE ACCOUNT
Signing up signifies that you have read and agree to the Terms of Service and our Privacy Policy.

Figure 12.1: *Heroku sign up form*

8. You will be using the Heroku CLI to perform the deployment from the command-line. Install the Heroku CLI using the following command:

   ```
   npm install -g heroku
   ```

9. This command will install the Heroku CLI globally so you can use it for all your future deployments.
10. To check the version of the Heroku CLI available, you can use the following command:

    ```
    heroku --version
    ```

When I checked it at my end, I got the following screenshot:

Figure 12.2: *Checking the Heroku version in the command line*

With Node/npm, git, and Heroku, we have all the prerequisites set up to get started with the deployment process.

For HTML/CSS/JS applications

There is one extra step needed for a simple JavaScript static application as there is no build or bundling process separately, and Heroku cannot detect the file to start the deployment with. This step will not be required when you perform the deployment for a React application in *Chapter 17, Debugging, Testing, and Deploying React Applications*.

If you do not perform this step and proceed with the hosting, you will get the following error:

No default language could be detected for this app.

> **HINT:** This occurs when Heroku cannot detect the buildpack to use for this application automatically.

In order to deploy the Notes app, we developed earlier, you would require to follow the given steps:

1. All your JS and CSS files should be placed in a public folder (as we did in *Chapter 9, Building an Application with JavaScript*).
2. Create a file called `composer.json` in the main project folder at the same level as `index.html` with the following content:

   ```
   {}
   ```
3. Create a file called index.php with the following line of code only:

   ```
   <?php include_once("index.html"); ?>
   ```
4. Adding this enables Heroku to detect your app as a PHP app and get started from there where it loads your `index.html`.

With these steps, now your JavaScript application is ready to be deployed.

Detailed steps for hosting

Now that you have signed up in Heroku and also downloaded the prerequisites, you are ready with your code bundle to be deployed.

Next, you will see the steps to host the application on the Heroku cloud platform. By following these steps from the terminal or command line, the process of deployment becomes very easy to accomplish.

Make sure you open the terminal in your project folder and get started with the following steps for deployment.

Step 1 – Login to the Heroku CLI

1. Use the following command to log in to the Heroku CLI:
   `heroku login`
2. You will be prompted with the following message:

```
heroku: Press any key to open up the browser to
login or q to exit.
```

3. Press a key and you will be redirected to the Heroku login page in your default browser as shown in the following screenshot:

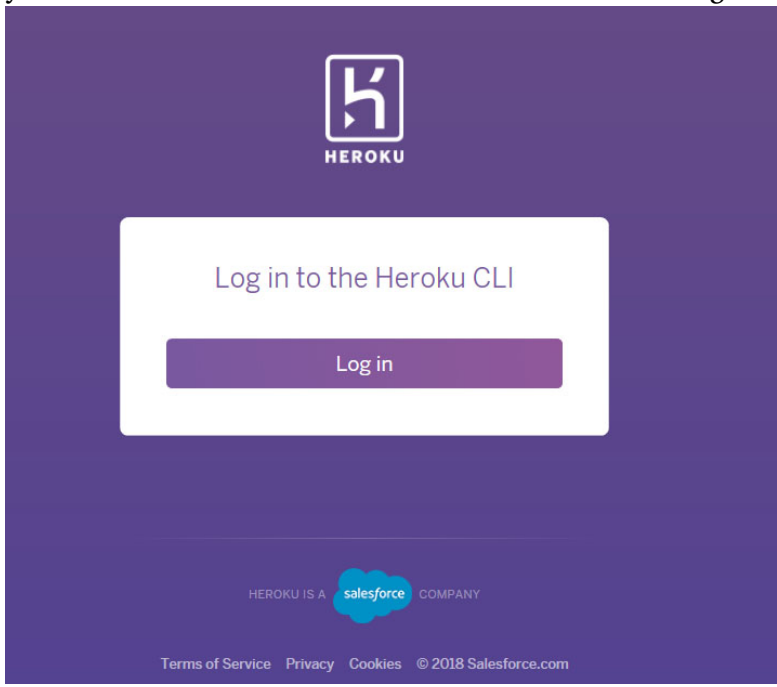

HEROKU
Log in to the Heroku CLI
Log in
HEROKU IS A salesforce COMPANY
Terms of Service Privacy Cookies © 2018 Salesforce.com

Figure 12.3: *Heroku login screen launched from terminal*

4. Log in to Heroku by providing your credentials and you will see the following confirmation screen:.

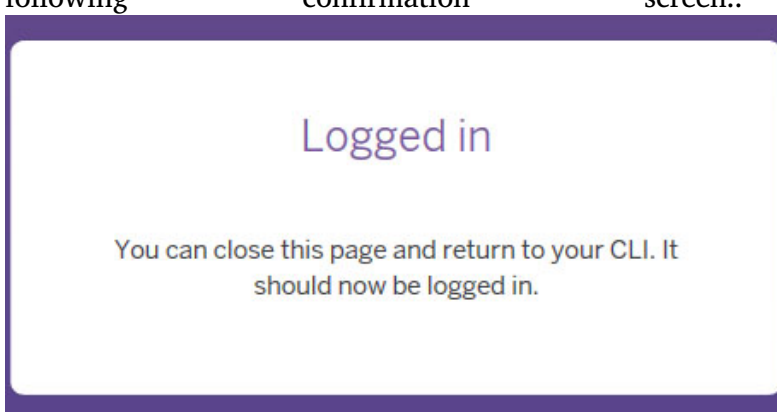


Figure 12.4: *Prompt to close the window after successful login*

5. After successful login, you can close the browser and come back to your terminal screen which is as follows:

```
heroku: Press any key to open up the browser to login or q to exit:
Opening browser to https://cli-auth.heroku.com/auth/browser/35f20962-fa45-4ac6-ab1d-3283a8f24bb2
Logging in... done
Logged in as abhilashahyd@gmail.com
```

Figure 12.5: Terminal after successful login

In case you get any error, close all browser tabs and retry the same command.

Alternate method:

If you want to stay in the CLI and enter the login details there itself, you can alternatively use the following command and you will be prompted to enter the details:

```
heroku login –i
```

Step 2 – Initialize and commit the application code to Git

The following steps are used to commit your code to Git:

1. Use the following commands to first initialize Git:

   ```
   git init
   ```
2. Add all the code to git using .:
   ```
   git add .
   ```
3. Commit the first version of the code using the following command:
   ```
   git commit –m "initial commit"
   ```

Step 3 – Create a named app in Heroku CLI

1. Use the following command to create an app on Heroku:

   ```
   heroku create mynotesapp4book
   ```

   The following screenshot shows the terminal after you enter the command to create an app on Heroku:

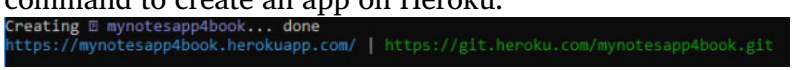
   

   Figure 12.6: Terminal when creating named app in Heroku CLI

2. The name given is just an example here. You will have to give a unique name to your app. If no name is given, Heroku gives a default name on its own like `thawing-oasis-43050`.
3. It creates and gives two sets of URLs as output:
   - The first URL is where the app will be deployed and available at **https://mynotesapp4book.herokuapp.com/**.
   - The second URL, **https://git.heroku.com/mynotesapp4book.git**, is the remote git repository URL where Heroku will create a git repository for your app code.

Step 4 – Push your code into the remote git repository

1. Use the following command to push the code into the remote git (the second URL from Step 3):

   ```
   git push heroku master
   ```

2. This will perform the final deployment where the code will be pushed into the git remote.
3. You will see a number of commands executed and a final confirmation of successful deployment.
4. This completes the process of deployment and now your application will be accessible on the first URL (**https://mynotesapp4book.herokuapp.com/**) from Step 3.
5. You can also use the following command from the Terminal to launch your application:

   ```
   heroku open
   ```

The deployment is complete and your application is now live on the internet for you to share it with your friends and colleagues.

Step 5 – Deployment of any changes made to the application

After the first deployment is done, there may be changes done to your application and every time you are ready with a verified and tested change, you would want to publish this new version onto the live production server.

Follow the given commands after connecting to the Heroku CLI to push the changes to the remote git:

1. Add the changed code to git:

   ```
   git add
   ```

2. Commit the changes with an appropriate message indicating the changes:

   ```
   git commit –m "The new set of changes for testing"
   ```

3. Finally, push the committed changes to the remote git:

   ```
   git push heroku master
   ```

This will make all your changes go live!

Alternate method

The first method was using the command line terminal. The alternate method is using the Heroku application online.

If you are more of a front-end person and do not like using the Terminal, you can do the deployment online using your account on the **heroku.com** website.

If you log in to your **heroku.com** account, you will be able to see all your apps which you have deployed:

Figure 12.7: *Heroku screen showing all your apps*

You can connect to your GitHub account where you maintain your code versions and directly deploy from there as well. You can navigate to the deploy tab in your Heroku account and sync your GitHub account with Heroku. Once connected to GitHub, you can sync your Heroku with GitHub and also set up for automatic deployment, whenever you push your code to the GitHub project master repository.

Using the terminal approach gives better control of the overall process and is also easy to follow with simple commands.

Conclusion

In this chapter, we learned about the process of deployment to make our application available on the internet for others to be able to access the same using a URL. Heroku is one of the cloud platforms which make the deployment process easy and free to try it out. There are other platforms which provide similar capabilities. We can try the deployment process for our own application and explore the platforms available.

Now that we have learned how to build and deploy JavaScript applications, we will be introduced to the JavaScript best practices in the next chapter.

Questions

1. Heroku is the only platform which is used for deploying JavaScript applications.

   A. FALSE

   B. TRUE

   Answer: Option A.

2. The following command is used to push the final changes to be available on Heroku remote git.

   A. git add

   B. heroku create

C. git push heroku master

D. git commit

Answer: Option C.

3. The heroku create command cannot be executed without providing a name and gives an error.

A. TRUE

B. FALSE

Answer: Option B.

4. Which of the following are the prerequisites used in this chapter for deployment?

A. Heroku CLI

B. npm/Node

C. git

D. All of the above

Answer: Option D.

5. The initial app and subsequent changes made to the application can be published to the app using?

A. Heroku CLI

B. Heroku application

C. Both A and B

D. None of the above

Answer: Option C.

CHAPTER 13

JavaScript Best Practices

"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live."

— John F. Woods

Software best practices lay down a set of rules and methodologies to be followed to ensure good quality of the end result. Following best practices are really not mandatory to learn coding and become a programmer, but this is what differentiates any code from beautiful, well-written, and performant code. There may be many ways to solve a problem, but there are some things to always keep in mind to write good code, no matter how you approach the solution. In this journey of yours to become a web developer, imbibing best practices will add more value to your learning as well as to your worth as a professional.

Structure

- JavaScript best practices
- HTML and CSS best practices

Objective

At the end of this chapter, you will get an insight into the best practices of programming in HTML, CSS, and JavaScript to write good code. However, the best practices keep evolving and if you can build a habit of continuously improving on the recommendations of this chapter, then that will give you true satisfaction as a professional.

JavaScript best practices

JavaScript best practices have evolved along with the language and continue to improvise. Some of the best practices you see in this section are general and hold good for programming in any language.

Readability

Readability is key to understanding someone's code and you must pay close attention to this aspect to ensure that others are able to understand your code. Above any best practice, I consider this at top because if someone is able to read your code well, then only he/she can make out rest of the things.

Naming conventions

- Always follow consistent naming conventions for the names
- For variables and functions, use self-explanatory, meaningful, and short names in CamelCase
- The global variables and constants (if you do need to use) shall be named in uppercase

The following example code shows sample names and whether they are good or not:

```
 let a,x,b, ghy; //bad names
 let count=0; // good name
 let firstName, lastName; // good name using camelcase

checkSignal(flag){ //good function name
 If flag===”red”
 return “stop”;
 else
return “go”;
}
```

Indentation and spacing

Proper indentation and spacing is key to a good overall code presentation:

- Always put one space after the operator and comma.
- Use 2 spaces or 4 spaces for indenting the nesting in your code. While I prefer 2 spaces, sometimes a team prefer 4 spaces. The point is that everyone in the team must use the same number of spaces for indentation.
- Keep the line length to be less than 80 characters.

Guidelines for objects

Some conventions should be followed while defining and using objects:

- Put the opening bracket { on the same line as the object name.
- Put the ending bracket } on the new and last line. Don't put anything else on that line, not event comments.

- Use a colon (:) immediately after the property name and give one space between the value and the colon.

Add sufficient comments as needed

Comments should add value and enhance the understanding provided by code. It should not be repetitive, that is, don't add comments for code statements which are clear by themselves.

In the following example, the function is simple and self-explanatory; hence, a comment may be unnecessary for such a self-explanatory code:

```
/*function to add 2 numbers and return the sum – This
is unnecessary for such a self-explanatory code*/
addNumbers(num1, num2){
return num1 + num2;
}
```

Comments are useful to explain why some part of the code exists, how some API are being used, or to explain results of some code action. Always keep the comments updated when the code is being changed. Out-of-date comments are even worse than no comments as they add to confusion instead of giving clarity.

Avoid global variables

Avoid use of global variables and references as those may get overwritten. Instead use ES6 modules and create closures to selectively expose only the required content.

In the following code snippet, a module is defined including some properties and methods. `getAge` and details are not exposed to the outside scope and cannot be accessed, only the `canVote` function is made available for access:

```
module = function(){
var details= {
    'name':'test',
    'age':'20'
  };
varcanVote= function(){
constcurrentAge = getAge();
    If (currentAge>=18)
return true;
  else
 return false;
  };
vargetAge= function(){
```

```
    return details.age;
  }
  return{canVote:canVote}
}();
module.canVote();
```

Follow strict coding style

Browsers are very adjusting with the JS code and do not throw errors for every issue.

To ensure that we maintain syntactical quality and catch any issues with the syntax, it is a good practice to validate your code using JSLint: **http://www.jslint.com/**.

JSLint is a debugger which scans the code and reports any problems with style conventions and code structure.

The use strict directive introduced in ECMAScript version 5 is used to indicate that the code needs to be executed in strict mode. With strict mode, JS becomes less lenient with its syntax; for example, you cannot use undeclared variables.

Write modular code – Single responsibility principle

Each function should perform only one task and fulfill only one responsibility. Writing modular functions which perform one specific task make them very reusable.

Bundle up related functions into modules which can be exposed selectively as APIs to be used across the application.

Always declare variables

If variables are not declared in JavaScript, they are considered as global, if you are not using the strict mode. We should always declare the variables to avoid global variables.

As we saw with ES6, there are now three ways to declare a variable:

- `var`: This is used for function level scoping and can change the value reference
- `let`: This is used for block level scoping and can change the value reference
- `const`: This is used for block level scoping but cannot change the value reference

It is always advisable to use the appropriate type based on the usage of the variables.

It is better to use the block scope as the variable is valid only within the block it is defined and not accessible anywhere in the function. This avoids any confusion with respect to the value being available and accessible outside of the scope it was declared.

The variables should be defined preferably as const or let, depending on their change behavior as follows:

- If the value does not change, prefer to use `const`
- If the value can change, use `let` instead
- Avoid use of `var`

Always initialize variables

Always initialize your variables with a default value to avoid any undefined errors and surprise values.

Avoid the use of `new()`, especially for the initialization of objects, arrays, and so on.

The preferred way of initializing variables depending on the data type is shown in *Table 13.1*:

Initialization	Datatype
Number = 0; (cannot be used)	Number
Boolean = false; (if false cannot be used)	Boolean
String = "";	String
Object = {};	Object
Array = [];	Array
Regular expressions = /(?:)/;	Regular expressions
Functions = function(){};	Functions

Table 13.1: *Datatype initializations*

Avoid heavy nesting

Avoid multiple levels of nested logic like a loop within a loop and within another loop. The code becomes complex to understand and the counters become difficult to maintain:

```
function renderNestedData(school){
for(vari=0;i<school.length;i++){
 console.log(school[i].details,'School');
var class= school[i].class;
for(var j=0;j<class.length;j++){
console.log(class[j].details,'Class');
var students= class[j].students;
for (var k=0; k<students.length; k++){
 console.log(students[k].personalDetails,'Personal');
 console.log(students[k].extraDetails,'Extra
```

```
 Curricular');
}
    }
   }
}
```

Instead of having a three-level nesting, it is better to define specialized methods to handle specific part of the tasks:

```
function renderNestedData(school){
for(vari=0;i<school.length;i++){
 console.log(school[i].details,'School');
var class= school[i].class;
displayClass(class);
   }
}

function displayClass(class){
for(var j=0;j<class.length;j++){
console.log(class[j].details,'Class');
var students= class[j].students;
displayStudent(students);
    }

function displayStudent(students){
for (var k=0; k<students.length; k++){
 console.log(students[k].personalDetails,'Personal');
 console.log(students[k].extraDetails,'Extra
 Curricular');
}
}
```

Optimize loop logic

Loops are iterative logic, which means the code inside the loop gets executed multiple times. So, we should be careful with what logic is being placed within the loop.

We need to check whether the logic is really required to be within the loop:

- Avoid variable declaration and initialization inside the loop.
- Avoid common computational logic in the loop.
- Avoid DOM manipulation in the loop.
- Use let instead of `var` to make the counter block-scoped rather than function-scoped:

```
for (let i = 0, len =numArr.length; i<len; ++i ) {
```

```
}
```

- Avoid any logic that changes the boundary condition of the loop resulting in infinite looping:

```
 let numArr = [1, 2, 3, 4, 5];
for (let i = 0, len = numArr.length; i<len; i++) {
numArr.push(i);
}
```

- Prefer for…of and Objects.key over for…in for optimized looping.

Use === Comparison

Avoid use of == instead of === for comparison.

=== is used for strict comparison that returns true only if both type and value are the same, whereas == does type coercion.

In type coercion, the two values are compared only after attempting to convert them into a common type:

```
5===’5’ //false since the datatype is different
5==’5’ // true
0 == ""; // true
1 == "1"; // true
1 == true; // true

0 === ""; // false
1 === "1"; // false
1 === true; // false
```

Similarly, !== is the strict non-equality operator and should be used instead of != for inequality checks.

However, == comparison exists for a reason. Hence, when you are sure that you are expecting a loose comparison and whether a value of number 10 or string 10 is not relevant as long as they are the same, then use this comparison consciously.

Differentiate between assignment and comparison

Many times, we end up using = instead of == or ===. This changes the value of the variable and introduces hard-to-detect bugs.

Differentiate between null and undefined

In JavaScript, the object can have a value as `null`, while they may be declared and defined. However, there are constructs like objects, variables, properties, methods, and so on, which may be undefined. In this case, testing whether they are empty or not could be tricky.

Use the following approach for testing the emptiness:

```
if (typeofobjToTest !== "undefined" &&objToTest !==
null) {
return true;
}
```

In a project environment, you need to define a utility function by wrapping the preceding logic to test the emptiness of an object.

Always use semicolon

Most browsers will not report any errors if you drop semicolons. JavaScript has an automatic semicolon insertion mechanism, but this may result in errors difficult to find by inserting semicolons at wrong places.

The `return` statement is a very common trap:

```
return
{
name:'New',
 Age:'18'
}
```

In the absence of an explicit semicolon at the end, the preceding code return will return undefined as the semicolon will be inserted on the first line of code after return.

This can be avoided by having the { immediately after the return keyword instead at the beginning of the next line:

```
return{
name:'New',
 Age:'18'
}
```

This convention should be followed for all blocks which are included in {} like if, for statements as it tells the browser that there is content following and it's not the end of statement:

```
if(num1===num2){
 return 2*num1;
}
for (i=0, i<length;i++){
result=result+1;
}
```

It is better practice to use semicolons at the end of statements (except function and class declarations).

Use default in switch statement

Always use ‘default’ in a switch statement. Even if you are not sure what you will do in the default block, it is important to explicitly think about this and handle this.

Differentiate between addition and concatenation

Depending on the context (that is, operands being used), the + operator acts as addition as well as concatenation. When you are not sure about the exact data type that you will get, it may be a good idea to first convert the operands into numbers using the function Number and then apply the addition operation.

Trailing comma in JSON and object definitions

A declaration like the following is dangerous in certain browsers and your application may crash:

```
friends = {"firstName":"Abhilasha", "lastName":"Sinha",
"gender":”Female”, }
```

Hence, make sure that you don’t have a training comma in the end.

HTML and CSS best practices

Here are some important coding practices to keep in mind when defining and styling your web content using HTML and CSS.

Maintain a clean and readable code structure

Readability is an important aspect in writing any code:

- Maintain clean, readable, and well-structured code by properly closing tags with end tags. To debug a broken page, first match all open tags with their closing tags.
- Use consistent naming convention.
- Use lowercase for tag names, attribute names, style properties, and values (except for specific uppercase values like file names, and so on).
- Use proper `DOCTYPE` that indicate the type of document like `HTML`, `XHTML`, and so on.
- Attribute values should be enclosed in quotes (single or double).
- Use comments to clearly identify a set of CSS rules.

Use the top-down approach in defining CSS in the similar order as the associated HTML elements appear on the page for ease of association and

understanding.

The following code shows a properly structured HTML code using lowercase tags:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Title</title>
</head>
<body>
<p>Well-structured Content using lower case tags </p>
<imgsrc=”IMAGE.jpg” alt=”My Image”/>
</body>
</html>
```

Include JS Script at the end

The JavaScript code is usually used to add some behavior to the page which may not be required at the start of page load and maybe linked with some user action. When the JavaScript code gets executed, it blocks the other part from being executed until it gets loaded completely. Hence, it is a good practice to place the script tags to include the JavaScript logic at the bottom of the HTML so that it does not slow down the HTML content load of the page.

Avoid inline styling

It is better to make use of external stylesheets to have all the CSS code segregated and well-defined instead of having it embedded in the HTML code in the form of inline styles. Inline styles are less flexible as they apply to a specific element. With external styles, the same set of styles can be applied to multiple pages which will be favorable for any website’s pages to have a consistent look and feel. Also, to make any change, the external CSS can be replaced or modified without touching the HTML code.

CSS to be used consciously

Some CSS selectors and properties can be expensive compared to others and should be used sparingly depending on the requirement at hand. Use these if you absolutely need them for your requirement:

- **!important:** This overrides the normal specificity and cascade and holds highest priority; hence, should be used sparingly.
- **descendant selector:** This is expensive as it checks for all the descendants.

- **position:** absolute/fixed: This breaks the normal flow of the elements, so it may become an overhead to maintain and handle multiple absolute/fixed positioned elements:
    - `border-radius`
    - `box-shadow`
    - `filter`
    - `:nth-child`
    - `transform`

Include imports wisely

Avoid unnecessary imports. Avoid import of unused links, images, scripts, and fonts and always make sure you load only what you are using. Always use the `link` tag to import external stylesheets as it works in parallel, instead of the `@import` directive which blocks parallel loads.

Use lists for navigation items

Lists are an effective way to present data in a structured and controlled format which is easy to modify and apply styling. Lists are ideal to create navigation menus and similar styled items.

The following is a list of navigation items implemented using an unordered list and anchor tags:

```
<ul class="nav-items">
<li><a href="#">Home</a></li>
<li><a href="#">Dashboard</a></li>
<li><a href="#">About Us</a></li>
<li><a href="#">Contact</a></li>
</ul>
```

Avoid unnecessary divs

When we start with HTML, it is common to keep adding the comfortable wrapper of div wherever we want to add some extra styling to the block and then again wrap around another div to just have a complete container. This tends to become a habit. When we re-analyze our code, we will surely find many unnecessary div tags. A wrapper div is not always required and the styling can be directly applied to any parent element. A wrapper may be required if you really don't have a containing tag where you want to apply some common styles to the sub sections.

Make sure to re-check and remove all such unnecessary container tags:

```
<div>
```

```
<ul class="nav-items">
<li><a href="#">Home</a></li>
<li><a href="#">Dashboard</a></li>
</ul>
</div>
```

Avoid tag qualify

Tag qualifying is the practice of associating extra element selectors with selectors which may be enough on their own.

Here, we are attaching myClass with the p tag:

```
p.myClass{
color : red;
text-align: center;
}
```

Do we really need a p tag here when the class is really enough to pick up the specific elements?

Similarly, attaching a tag or class to an ID selector will have the same impact:

```
div#myID{
Background: blue;
}
```

If you investigate the behavior of browsers closely and understand how they read your CSS selectors, you will get to know that they parse the selectors from right to left. That means that in the selector `ul> li a[title="play"]`, the first thing being interpreted is `a[title="play"]`. This first part is also referred to as the key selector which ultimately gets selected.

Whenever you are defining your selectors, be careful if your right-most selector is self-sufficient; do not over qualify them unnecessarily with extra selectors, as this reduces the overall selector's efficiency.

Use color hex codes

I prefer the use of color hexadecimal code instead of the color names as they are slightly faster:

```
.myClass{
color : #ff0000;
}
```

Consider cross-browser compatibility

Use suitable CSS vendor prefixes when using a new feature of CSS3 to make sure that your code is supported by most browsers. When using something new, you can check the available browser support @mailto:@**https://caniuse.com/**and check whether you need to prefix @ **http://shouldiprefix.com/**.

Validate your code

Validate your HTML and CSS code using W3C free CSS Validator (**https://jigsaw.w3.org/css-validator/**) which will perform quality checks and let you know if your web page is properly structured.

Reduce CSS code size

Write short, effective code which is easier to download and parse.

Make use of shorthand properties wherever applicable. Avoid repetitive code and clean up unused code. Group your selectors wherever common rules apply.

You can minify your CSS file using tools like CSS Compressor and Minifier to reduce the final file size.

Recommendations on best practices

In a book on JavaScript, it is always challenging to accommodate all the good practices that you would like to see in a real project like environment. However, it is important to have guidelines available and easily accessible for everyone in the team. The guidelines must be initially derived from a reliable source (for example, this book or anywhere else) and as a team you must continue improving the guidelines by incorporating your learning from projects and various other sources. In addition to the best practices in this chapter, also visit the following URLs to develop a deeper understanding:

- **https://dev.opera.com/articles/javascript-best-practices/**
- **https://github.com/airbnb/javascript**

Further, most likely you will be using JavaScript with certain UI development frameworks like React.js, Angular, Ext JS, Vue, and so on. So, it is important that you also go through the best practices of the framework being used in your project. The lessons discussed in this chapter are generic and it shall be applicable across the frameworks; however, whenever in conflict, go with the recommendation for the framework so that other developers find it easier to make use of your code.

As discussed in the beginning of the chapter, the quality of the code is what differentiates a novice from a professional. We believe that if you

have the passion to write the best quality code, then programming looks like a science wrapped inside the most precious art. Otherwise, you will just end up draining yourself in the pursuit of finishing the tasks and most likely the end of creating great technical debts.

Also, we often hear that following coding guidelines will take longer to finish a task and due to timeline pressure, people skip the coding guidelines. We consider that as a hygiene issue. Build a habit to write the best code and you will be guaranteed to save time and hopefully you will have lesser timeline pressure.

Conclusion

By following best practices and writing good quality code from the beginning of your career, you will be able to establish yourself as a true professional who is respected and looked up to by others. Being consistent and committed to good coding practices becomes a priceless programming habit. After learning how to write good JavaScript code, in the next chapter, we will learn about one of the popular front-end JavaScript frameworks, ReactJS.

Questions

1. _____ is the practice of associating extra element selectors with selectors which may be enough on their own.

   A. Tagging
   B. Extra Taggify
   C. Tag qualifying
   D. None of the above

   Answer: Option C.

   Tag qualifying is the practice of associating extra element selectors with selectors which may be enough on their own and should be avoided.

2. This overrides the normal specificity and cascade and holds the highest priority, and hence, should be used sparingly. What is this?

   A. CSS Id selector
   B. !important
   C. CSS z-index
   D. position property

   Answer: Option B.

   !important option applied to any rule overrides the normal specificity

and cascade and holds highest priority, and hence, should be used sparingly.

3. If variables are not declared in JavaScript, they are considered as global, if using the Strict mode.

   A. TRUE

   B. FALSE

   Answer: Option A.

   If variables are not declared in JavaScript, they are considered as global, if they are not using the Strict mode. If they are using the Strict mode, this gives an error.

4. ____ is used for strict comparisons and should always be used in this form.

   A. = = =

   B. = =

   C. =

   D. None of the above

   Answer: Option A.

   = = = is for strict comparison that returns true only if both type and value are the same, whereas = = does type coercion.

5. Which of the following is an incorrect coding best practice?

   A. Avoid tag qualify in CSS

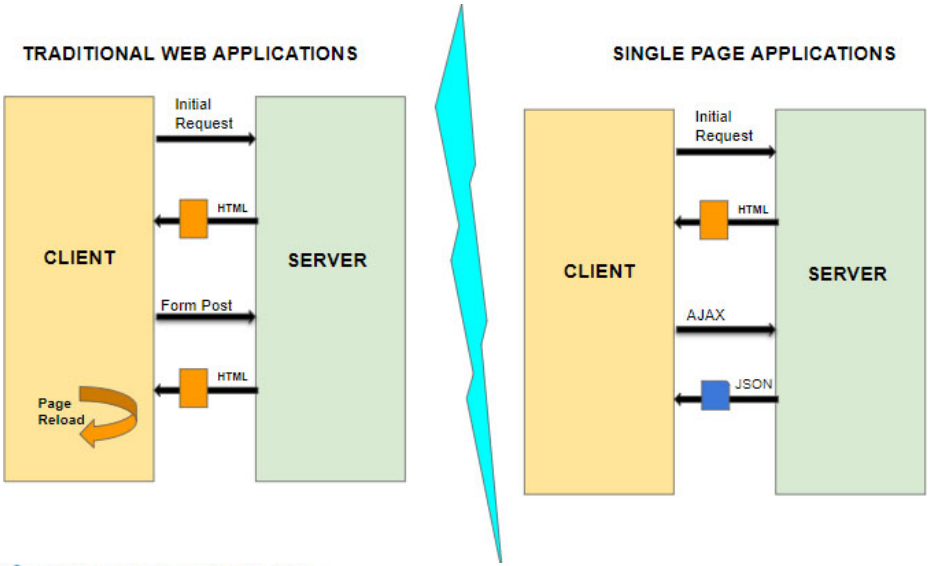
   B. Avoid Heavy Nesting logic in JavaScript

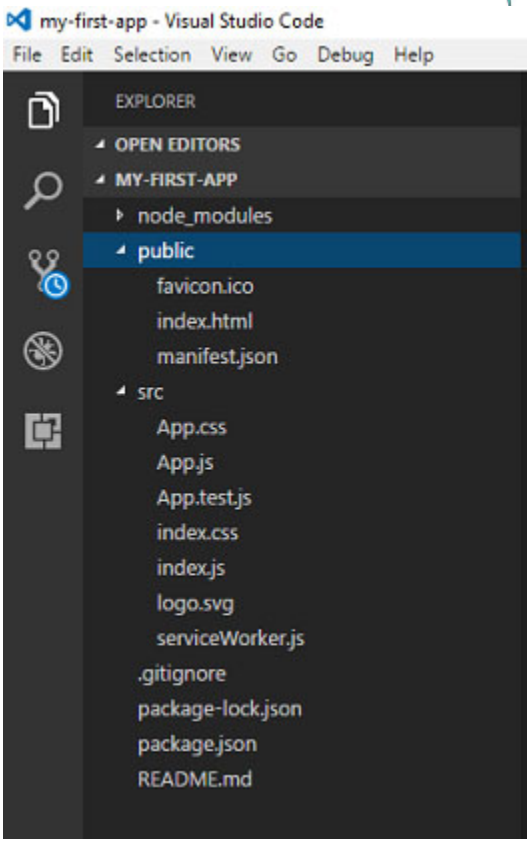
   C. Use default in switch statement

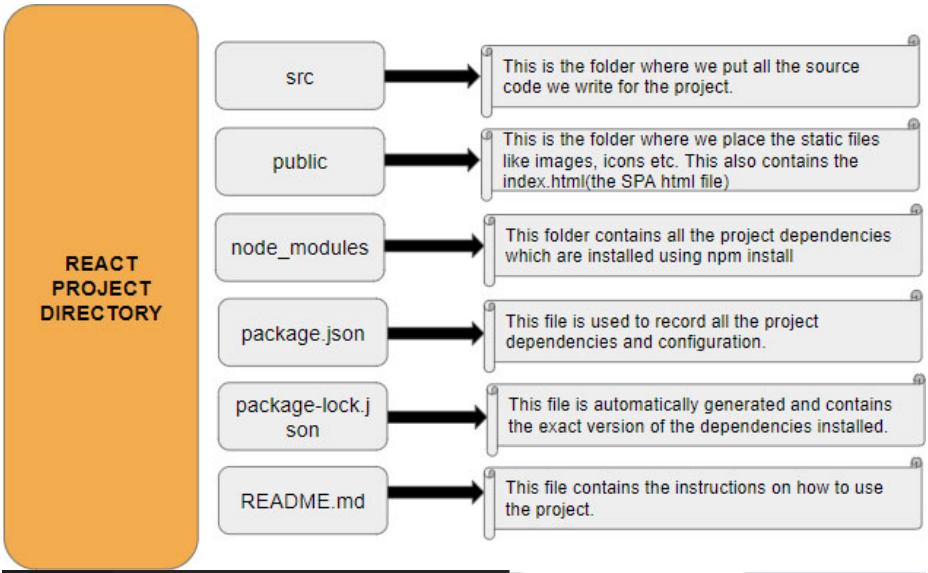
   D. Avoid semicolon as adding it is not mandatory

   Answer: Option D.

   Always use a semicolon. Most browsers will not report any errors if you drop semicolons. JavaScript has an automatic semicolon insertion mechanism, but this may result in errors difficult to find by inserting semicolons at wrong places, so always use a semicolon.

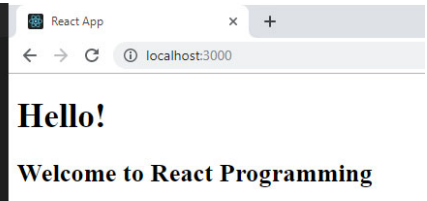

TRADITIONAL WEB APPLICATIONS
Initial Request
HTML
CLIENT
SERVER
Form Post
HTML
Page Reload
SINGLE PAGE APPLICATIONS
Initial Request
HTML
CLIENT
SERVER
AJAX
JSON


my-first-app - Visual Studio Code
File Edit Selection View Go Debug Help
EXPLORER
OPEN EDITORS
MY-FIRST-APP
node_modules
public
favicon.ico
index.html
manifest.json
src
App.css
App.js
App.test.js
index.css
index.js
logo.svg
serviceWorker.js
.gitignore
package-lock.json
package.json
README.md


REACT PROJECT DIRECTORY
src
This is the folder where we put all the source code we write for the project.
public
This is the folder where we place the static files like images, icons etc. This also contains the index.html(the SPA html file)
node_modules
This folder contains all the project dependencies which are installed using npm install
package.json
This file is used to record all the project dependencies and configuration.
package-lock.json
This file is automatically generated and contains the exact version of the dependencies installed.
README.md
This file contains the instructions on how to use the project.

index.js

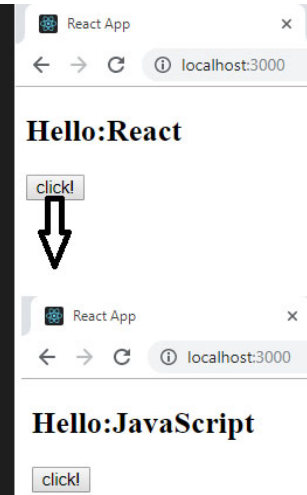
```
import React from 'react';
import ReactDOM from 'react-dom';

const element = (
  <div>
    <h1>Hello!</h1>
    <h2>Welcome to React Programming</h2>
  </div>
);

ReactDOM.render(element, document.getElementById('root'));

```

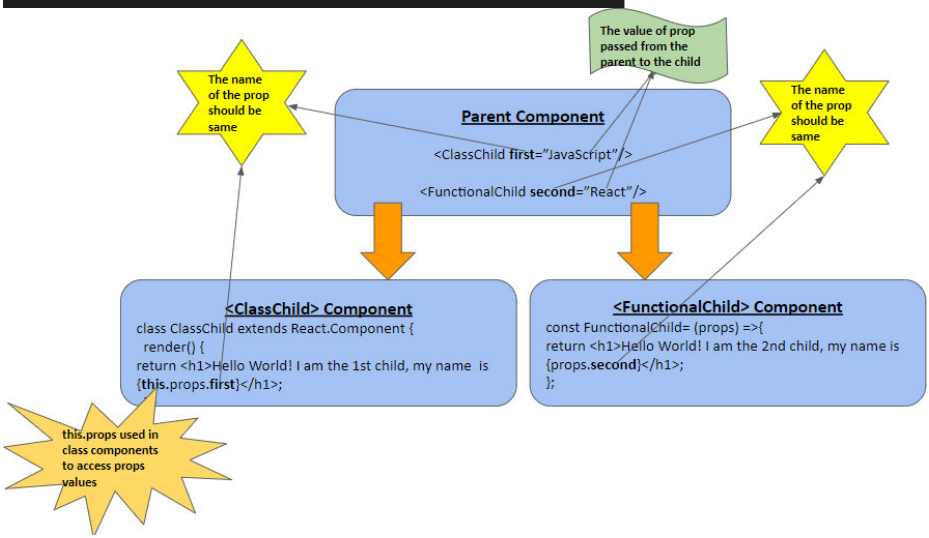



```
var helloWorld= (
  <div>
    <h1> Hello World </h1>
    <h2>Welcome to Modern Web app development</h2>
    <p> React is fun, Let's get started ! We will learn about
      <span> JSX,
        <b>components</b>,
        <em>state</em> and
        <i>props</i>
      </span>
    </p>
  </div>
);
```

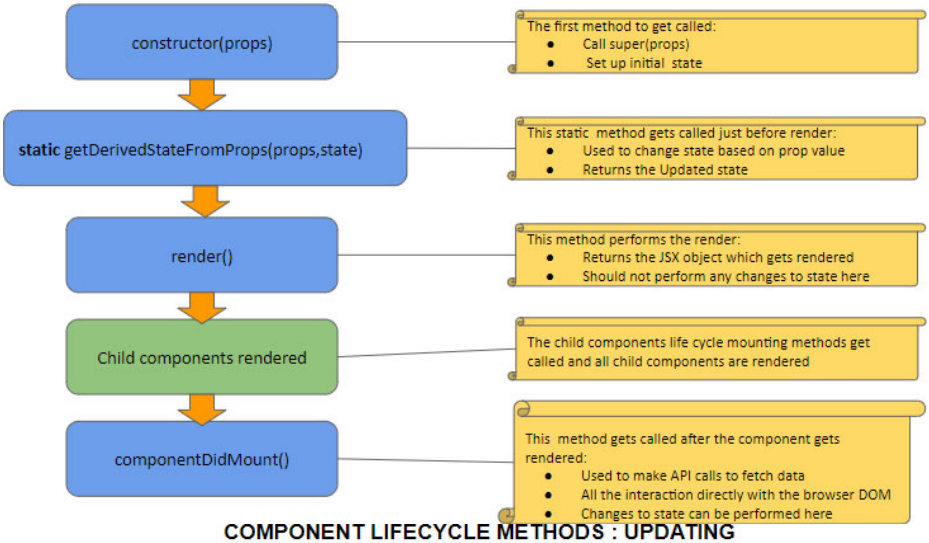
```
"use strict";

var helloWorld = React.createElement("div", null,
React.createElement("h1", null, " Hello World "),
React.createElement("h2", null, "Welcome to Modern Web app
development"), React.createElement("p", null, " React is fun,
Let's get started ! We will learn about",
React.createElement("span", null, " JSX,",
React.createElement("b", null, "components"), ",",
React.createElement("em", null, "state"), " and",
React.createElement("i", null, "props"))));
```

```jsx
class NameChange extends React.Component {
 state = { Name: "React" };

onNameChange = () => {
  if(this.state.Name==="React"){
   this.setState({
     Name: "JavaScript"
   })
  }else{
    this.setState({
      Name: "React"
    })
  }
}
  render() {
    return (
      <div>
        <h2>Hello:{this.state.Name}</h2>
        <button onClick={this.onNameChange}>click!</button>
      </div>
    );
  }
};

ReactDOM.render(<NameChange />, document.getElementById('root'));
```

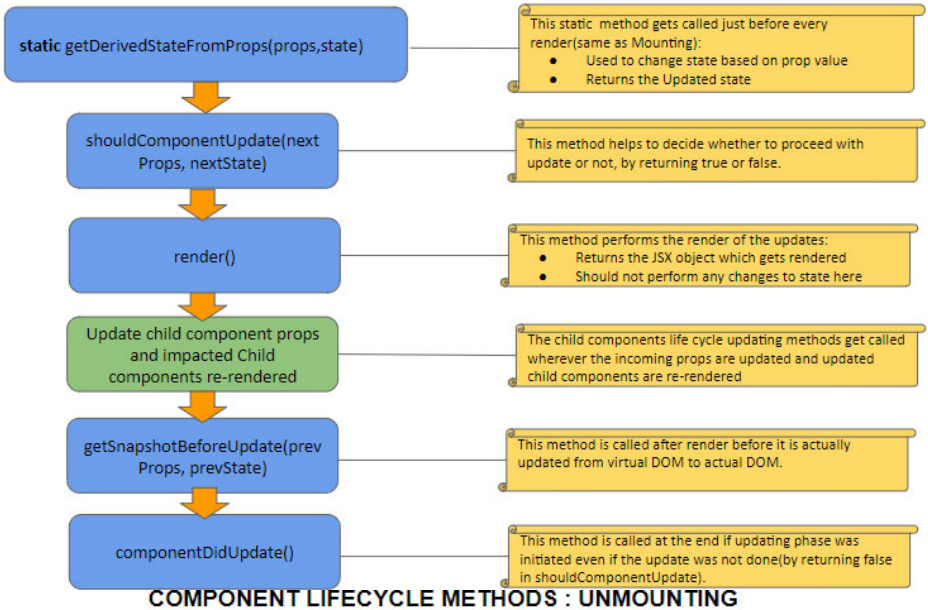




COMPONENT LIFECYCLE METHODS : MOUNTING

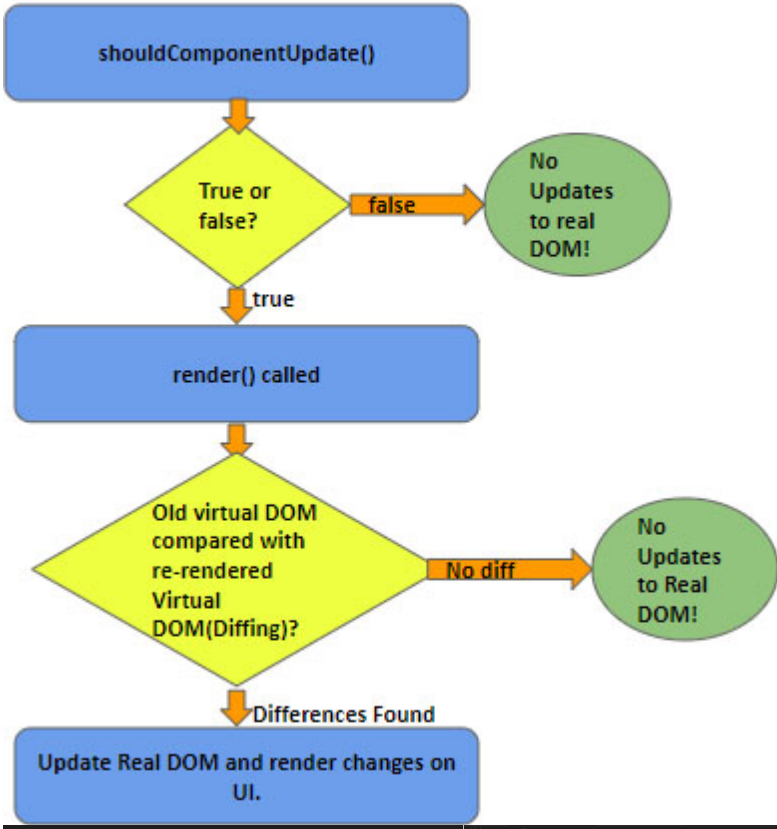

constructor(props)
The first method to get called:
Call super(props)
Set up initial state
static getDerivedStateFromProps(props,state)
This static method gets called just before render:
Used to change state based on prop value
Returns the Updated state
render()
This method performs the render:
Returns the JSX object which gets rendered
Should not perform any changes to state here
Child components rendered
The child components life cycle mounting methods get called and all child components are rendered
componentDidMount()
This method gets called after the component gets rendered:
Used to make API calls to fetch data
All the interaction directly with the browser DOM
Changes to state can be performed here

COMPONENT LIFECYCLE METHODS : UPDATING


static getDerivedStateFromProps(props,state)
This static method gets called just before every render(same as Mounting):
Used to change state based on prop value
Returns the Updated state
shouldComponentUpdate(next Props, nextState)
This method helps to decide whether to proceed with update or not, by returning true or false.
render()
This method performs the render of the updates:
Returns the JSX object which gets rendered
Should not perform any changes to state here
Update child component props and impacted Child components re-rendered
The child components life cycle updating methods get called wherever the incoming props are updated and updated child components are re-rendered
getSnapshotBeforeUpdate(prev Props, prevState)
This method is called after render before it is actually updated from virtual DOM to actual DOM.
componentDidUpdate()
This method is called at the end if updating phase was initiated even if the update was not done(by returning false in shouldComponentUpdate).

COMPONENT LIFECYCLE METHODS : UNMOUNTING

How Virtual DOM works?




```
//index.js
class CourseList extends React.Component {
 state = {
   courses:[
   {name: "React", prereq: "JavaScript,CSS,HTML" },
   {name: "JavaScript", prereq: "None" },
   {name: "HTML", prereq: "None" },
   {name: "CSS", prereq: "None" }
 ]};

nameChangeHandler(event,index){
  let courses =[...this.state.courses];
  courses[index].name= event.target.value;
  console.log(courses);
  this.setState(courses);
}

 render() {
    return (
      <div>
        <h2>Hello!</h2>
       { this.state.courses.map((course, index)=>{
           return <Course key={index} name={course.name} prereq={course.prereq}
           click={(e)=>this.nameChangeHandler(e,index)}/>
        })}
      </div>
    );
  }
};

ReactDOM.render(<CourseList />, document.getElementById('root'));
```

```
//Course.js
import React from 'react';
import './index.css';

const course = ( props ) => {
    return (
        <div className="course">
            <h2>Course: {props.name}</h2>
            <p>Prerequisites: {props.prereq}</p>
            <input type="text" onChange={props.click} value={props.name} />
        </div>
    )
};
export default course;
```

```
/*index.css*/
.course {
    width: 40%;
    margin: auto 0px;
    border: 1px solid #eee;
    box-shadow: 0 2px 3px #ccc;
    padding: 16px;
    text-align: center;
    background-color: lightblue;
}
```

Hello!

Course: React

Prerequisites: JavaScript,CSS,HTML

React

Course: JavaScript

Prerequisites: None

JavaScript

Course: HTML

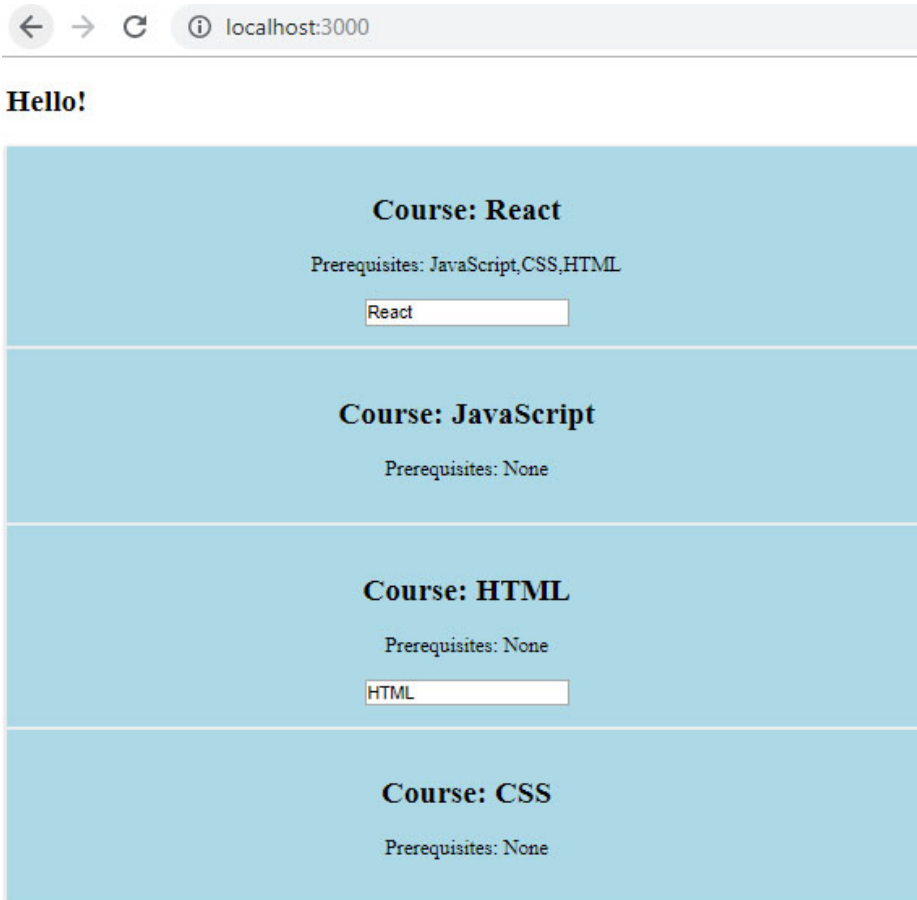
Prerequisites: None

HTML

Course: CSS

Prerequisites: None

CSS



CHAPTER 14

Introduction to React

'Don't bother just to be better than your contemporaries or predecessors. Try to be better than yourself.'

- William Faulkner.

In the previous chapters, we learnt how to build a web application using the basic web development skills, which include HTML, CSS and JavaScript. While this knowledge is good enough to get started in the field of web development, modern web development is much more than the vanilla JavaScript. With so many frameworks available today, there is a lot which can be done on this learning path. Frameworks/libraries avoid

repetitive code and provide an overall structure to the complete application. There are many popular frameworks like Angular, React, Vue, Ext JS, and so on, and they each can have an entire book about them. These frameworks help in building interactive and intuitive **single-page applications (SPA)** which helps in ensuring excellent **user experience (UX).** In this chapter, you will be introduced to one of the most popular frameworks, React and its main concepts. In the later chapters, we will cover the complete web development lifecycle of a React application to help you get started with this highly efficient and flexible framework.

Structure

- What are Single Page Applications (SPA)?
- Getting started with React
- Introduction to JSX
- Elements, components and props
- State, lifecycle methods and virtual DOM
- Passing method reference
- Styling components
- Rendering lists and conditionals
- Forms
- Composition
- Hooks

Objective

After completing this chapter, you will be aware of the concepts of React and how they work together to build highly performant applications.

Single Page Applications(SPA)

In traditional web applications, every request would be submitted to the server, which would send another HTML page as a response. So the application would navigate from one page to the other and multiple HTML pages would be loaded as the user interaction happens. It is similar to the application we built in *Chapter 9,* wherein we had two HTML files between which the application moved.

SPA are web applications which load a single HTML page (`index.html`), and parts of the same page get updated as the user interacts with the applications to give a seamless experience to the user. Data is fetched by AJAX requests not interrupting the application and updating parts of the application with the response received. The user gets the feeling that the

application is always available as there are no reloads and no intermediate waits. The overall application is faster and much more responsive. The following diagram shows the difference between traditional web applications and single-page applications:

Figure 14.1: *Traditional applications versus Single Page applications*

Getting started with React

React is a JavaScript library developed and used by Facebook. It deals with the view layer and helps in building highly interactive and performant single-page applications.

In this chapter, we will dive deeper and understand the concepts of React. Let's get started by building our first React app.

You may ask, how can we do that without even knowing anything about React. You can do it because of two reasons:

- React is not a new technology. It's JavaScript, which you already know now. Also, React uses more of ES6 syntax which we had seen earlier, so you may like to review that. There are ways to write the same code in old JavaScript syntax, but in this book, we will follow the ES6 syntax as that is the latest and more popular one. Additionally, there are some new concepts, terms and terminologies related to React, which we will learn here, so now is the time to get started.
- Building a template SPA in React is as simple as executing a set of commands on the command line.

So Let's begin. Open your command prompt.

Prerequisites: Latest NodeJS version should be downloaded from **https://nodejs.org/en/download/**.

We will install a tool for creating React apps. Aptly named as *create-react-app*, the tool scaffolds the file structure for your React app and includes a dev server, as well as a compiler (Babel), bundler (Webpack), and more, all the requirements to get us started with React.

This can be done in the two steps:

1. Build the app using the tool `create-react-app`, giving the application a proper name, which should not contain capital letters:

   ```
   npx create-react-app my-first-app
   ```

   This creates a complete folder with the name as the app name given, with a well-defined folder structure for a full-fledged React application. You can move into your project folder using -:

```
cd my-first-app
```

You can start the application using the given command from within the application folder:

```
npm start
```

The application launches in your default browser on `localhost:3000`.

You can view the default application by visiting `localhost:3000` on your browser. Our first React application looks like the following screenshot:

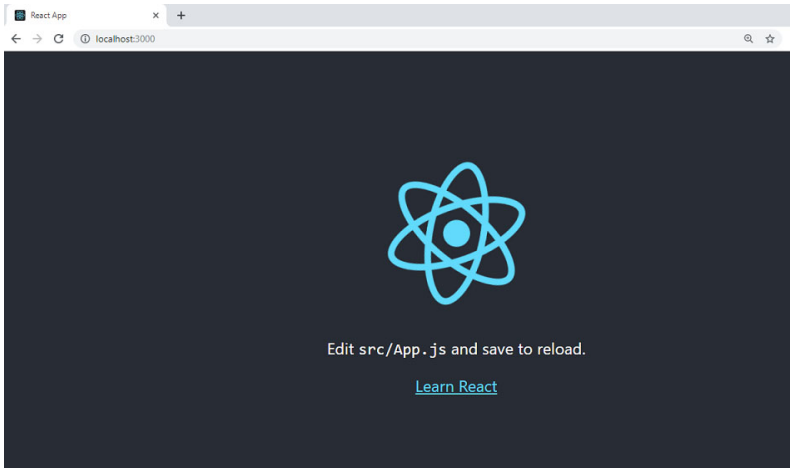


Figure 14.2: *My first React application using create-react-app*

To stop the application, use *Ctrl + C* on command prompt.

Now that we have built our first template app, let's look into the project's folder structure and understand the purpose of the structure. We will open the folder inside the editor and look at the files, as shown in *Figure 14.3:*

Figure 14.3: *React project folder structure*

The following diagram shows the important files in the structure and their purpose:

Figure 14.4: *Project folder/files and their purpose*

Before we get into details of React, there are a few basic things which we need to understand at this point.

src

The folder which contains all the application code which we write. In `my-first-app` project folder, you will see many files here. To understand, you need to look into the `.js` files, which are the react component code. The `.css` extension files are for styling to define corresponding classes to

style the react components. The mandatory file in this folder is index.js which the application looks for and which contains the code to render the React component into the DOM. You can delete all the other files in the src folder and only keep `index.js` to start with.

Let's understand the main part of the code of index.js:

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
ReactDOM.render(<App />,
document.getElementById('root'));
```

Here the App component is rendered into the DOM. This is defined separately in the `App.js` file and imported here.

The `index.html` file in public folder has the following line in the body part which marks the DOM element to which the React component is being rendered:

```
<div id="root"></div>
```

React and React-DOM

What are these two files which we import in a React App, included in index.js?

React is split into two separate libraries:

- `React`: This library knows what a component is and how to make components work together
- `React-DOM`: This library knows how to take a component and render it on the DOM

Now we understand how these two libraries include the functionality required to define components and then render them on the DOM.

Converting the template create-react-app into a simple Hello world app looks like this:

Figure 14.5: *Hello world code(index.js) and browser view*

You must be wondering looking at this code, as looks like JavaScript but it has some HTML like tags embedded into it. We will delve deeper and understand what this is and how this works.

In the following sections, we will learn about the key concepts of React, which include:

- JSX
- Elements

- Components
- Props
- State

Introduction to JSX

JSX is a syntactic extension of JavaScript which was written to be used with React. Its purpose is to be able to embed UI components comprising of HTML with CSS within JavaScript in a user-friendly syntax. That's the reason why it looks like HTML to define the view.

If a JavaScript file contains any JSX code, then the browser cannot understand the file directly. The file will have to be compiled into regular JavaScript before the file reaches a web browser. In order to see how your JSX code gets transpiled to plain JavaScript, you can check it by typing your code at **http://babeljs.io/repl**. Babel is the transpiler which does the transpilation from JSX to JS, which comes bundled when we set up our app using the create-react-app tool.

React can be written without using JSX but JSX makes the code much more compact, clear and easy to understand; hence it is highly adopted in React applications.

A basic unit of JSX is called a **JSX element**, which looks exactly like HTML but is found in a JavaScript file. This is what is used to define the view of the React application.

JSX elements are like JavaScript expressions. They can go anywhere like any other JavaScript expression, that is as follows:

- A JSX element can be saved in a variable
- It can be passed to a function as a parameter
- It can be stored in an object or array
- Numeric, string manipulations, direct variables or function calls

Basically, by using JSX you can write concise HTML/XML-like structures (for example, DOM like tree structures) in the same file as you write JavaScript code, then Babel will transform these expressions into actual JavaScript code.

By using JSX, the following JSX/JavaScript code can be written:

```
var helloWorld= (
<h1> Hello World </h1>
);
```

And Babel will transform it into this:

```
var helloWorld = React.createElement("h1", null, "
```

```
Hello World ");
```

When React creates elements, it calls `React.createElement()` method, which takes three parameters as follows:

- The element name (the tag, for standard tags it should be used in lowercase only. PascalCase is interpreted as custom components)
- An object representing the element's props
- An array of the element's children(what comes inside the element tree)

Just imagine how this will look if there are more tags, as shown in the following diagram:

Figure 14.6: *JSX code with its transpiled JS code in Babel*

The size will keep on growing as you add more tags! You can think of JSX as a shorthand for calling `React.createElement()` which acts as a syntactic saviour and saves us from writing these many, many lines of confusing code.

Let us now understand the JSX syntax.

- JSX can comprise of single line elements using HTML tags with embedded JS expressions enclosed within parentheses {}.

  ```
  const element = <h1>Hello, {name}</h1>;
  ```

  The variable element is assigned a JSX expression which includes some HTML tags. Also embedded is a JavaScript variable name, enclosed in parentheses.

  Whenever we need to embed JS code within JSX, we can do that within a pair of parentheses.

- JSX expressions can include any kind of expressions, numeric, string manipulations, direct variables or function calls:

  ```
  const element = <h1>Hello, 911!
  {ConvertToCaps(name)}</h1>;
  ```

  `ConvertToCaps(name)` JS function is called within the JSX within {}.

- If the JSX expression extends to multiple lines, it needs to be enclosed in round parentheses ():

  ```
  const element = (
  <div>
  <h1>Hello!</h1>
  <h2>Welcome to React Programming</h2>
  </div>
  );
  ```

- JSX expressions can be included in expressions like arguments or returned as results from functions:

```
if(showFlag)
return <h3>Flag is set<h3>;
else
return null;.
```

- JSX expressions should use double quotes for string values to be assigned to HTML tag attributes or braces to include JS expressions.
- if…else statements or loops cannot be directly included as part of JSX as they are statements and not expressions.

  JS conditional(ternary) operator ?:, can be embedded in JSX to achieve conditional rendering shown as follows as the JSX which gets returned depends on the value of showFlag:

```
 return (
<div>
  Show the flag {showFlag ? <Flag/> :"No Flag"}
</div>
 );
}
```

- JS logical operators && can also be used to achieve a similar effect of conditional rendering:

```
 return (
<div>
  showFlag &&<Flag/>
  !showFlag && "No Flag"
</div>
 );
}
```

  && condition evaluates to true only if both expressions are true, so only one of the two parts will get rendered.

- JSX can include any level of nested tags, but the complete expression should return a single root tag. The entire JSX expression should always have one root level tag to enclose the entire JSX content.

The view will be created using a single JSX expression or a combination of multiple JSX expressions. JSX forms an integral part of React programming, and it's easy to learn due to its similarity to HTML.

Adding event handlers

Event handlers can be added to handle specific events or user interaction. The event handler function should include the function name, not followed by braces only to pass a reference to the function and not invoke it.

It should be associated as the following to the action which needs to be handled:

```
<button onClick={this.onNameChange}>click!</button>
```

In order for `this` to work within the function, it is recommended to use `arrow` function for function declaration as follows:

```
class NameChange extends React.Component {
 state = { Name: "React" };

onNameChange = () => {
 if(this.state.Name===”React”){
  this.setState({
   Name: "JavaScript"
  })
 }else{
   this.setState({
    Name: ”React”
   })
 }
}
 render() {
  return (
<div>
<h2>Hello:{this.state.Name}</h2>
<button onClick={this.onNameChange}>click!</button>
</div>
  );
 }
};
render(<NameChange />,
document.getElementById('root'));
```

The preceding code looks like the following in action:

Figure 14.7: *Event handler to handle the button click event*

Elements, components and props

The core building blocks of React include elements and components.

Elements

React elements are the simplest building blocks and describe what is finally rendered on the screen. Each element corresponds to a DOM element on the screen, which is created using `React.createElement`. It's an object that virtually describes the DOM nodes that a component

represents. It is an immutable object which cannot be changed, but every time a new copy will be created.

How is an element rendered on the DOM? How is it associated?

In our `index.html`, we will have a `<div>` marked with a unique ID:

```
<div id="root"></div>
```

Now we can add any React element to the DOM using `ReactDOM.render` which takes two parameters: the element to be rendered, and the DOM node where the element is to be rendered, as shown below.

```
const element = <h1>Hello, world</h1>;
ReactDOM.render(element,
document.getElementById('root'));
```

Here, the element is the React element which contains the JSX code which the element represents.

`ReactDOM.render` is the standard function, part of the `react-dom` package, which does the rendering of the element to the single HTML page of our SPA, that is, index.html:

```
ReactDOM.render(<what to render>, <where to render>)
```

The first parameter is the React element to be rendered.

The second parameter is the DOM element from the `index.html` to which the JSX element will be rendered.

React elements are only plain old JavaScript objects which describe the HTML for the component. Elements are what gets returned by the `React.createElement()` method we had seen earlier, which is what the JSX code gets transpiled into.

Components

Components are the main building blocks of a React application which make up the UI. A React UI is made up of multiple independent React Component instances. Each Component renders some part of the UI returned in the form of a JSX expression.

Components form reusable pieces of UI, handled one at a time and put together to build complex UIs.

As a part of `ReactDOM.render`, what gets rendered on to the DOM is an instance of the component. A React component is a template or a blueprint which details out how the elements will be.

Then what gets rendered or returned from the component? It is the React element, which describes the DOM nodes in the form of an object.

React internally creates, updates, and destroys instances of components to form the DOM elements tree that needs to be rendered to the browser.

Always start the component names with a capital letter. The components starting with lowercase letters are considered as the standard DOM tags. Components starting with a capital letter are considered as custom components and React looks up for the component definition even if you use a standard tag with a capital letter(like `<Div></Div>`), and returns an error.

There are two types of React components, based on the way they are defined:

- Functional components
- Class components

Functional components

Functional components are the simplest components as they are just plain JavaScript functions which take a parameter called props and returns the JSX code, which represents the element to be rendered in the DOM.

The simplest `HelloWorld` way of defining a functional component is as follows.

This is using the ES6 arrow function and not making use of any props, only returning a fixed string:

```
const Hello= () =><h1>Hello World!</h1>;
```

Or in multi-line syntax with explicit return statement as follows:

```
const Hello= () =>{
return <h1>Hello World!</h1>;
};
export default Hello;// Exporting is required for other
components to be able to import it.
```

If you have to include the above Hello component in your App component or any other component, you can easily import the file and include the tag in the JSX of the container component as follows:

```
import Hello from "./Hello";

const App=()=>{
return(<div>
<Hello/>
</div>);
}
```

If we include props in the above definition as shown:

```
const Hello= (props) =>{
return <h1>Hello World! My Name is {props.name}</h1>;
};
```

Now the component expects a prop called `name`. The prop value should be provided for the included component to work as expected as shown below:

```
const App=()=>{
return(<div>
<Hello name="JavaScript"/>
</div>);
}
```

The preceding example is a very simple, functional component. Functional components are easier and sleeker to write and maintain, as they are only functions. They can handle logic within the function and finally return the JSX to be rendered.

Function-based React elements do not have instances as they are created by a functional component.

A function component does not have instances it is associated with whatever is returned as a result of the function invocation. It can be rendered multiple times, but there will not be any local instance associated with it for each render. It determines what DOM element to render for that function based on what is returned by the function.

In earlier versions of React, functional components were mainly for presentation with minimal logic as they have no state of their own and also have no lifecycle methods, like class components.

With the arrival of React Hooks in React v16.8 onwards, Functional components have become equally capable in terms of handling state as well as lifecycle methods. We will explore this in detail under Hooks.

With equal power, functional components will become the preferred method as it is much sleeker and does not have the additional overhead of managing class instances with methods.

Class components

Class components are defined using ES6 classes. It has a render method which returns the JSX that gets rendered on the UI as a part of that component.

For the earlier `HelloWorld` example, the class component will look like as follows:

```
import React from 'react';
class App extends React.Component {
 render() {
```

```
  return <h1>Hello World</h1>
 }
}
```

A class component must have the following:

- Must be a JS class
- Must extend `React.Component`
- Must define a render method to return some JSX

Props can be accessed in class components using `this.props`:

```
class Welcome extends React.Component {
 render() {
return <h1>Hello World! My Name is {this.props.name}</
h1>;
 } }
```

Stateful components and stateless components

There is another way of categorizing components into stateful components and stateless components. The characteristic is simple as its name suggests: the components that have state and the components that don't.

Stateful components are class components and functional components with hooks which can handle state. Stateful components can utilize the features of state and lifecycle methods. We will read more about how a state can be defined and used in stateful components in a later section of this chapter. Asynchronous functions cannot be used in stateless components and can only be handled in stateful components.

Stateless components are components which do not have any state associated with them and mainly used for presentation purposes only. It is usually recommended to use functional components as stateless as they are simpler to use.

Props

Props are properties or attributes of the component. These are used to pass dynamic values to the component from its parent. A **component** (functional or class) should not modify its props. Such a component is considered to be pure. React components are like pure functions for their props; they should not modify the props. The following figure depicts how props can be used with functional as well as class components:

Figure 14.8: *How Props are used in class and functional components*

The prop is passed as an attribute on the component tag when it is included in the JSX body of the parent. The name of the attribute used

there becomes the name of the prop, using which the value can be accessed inside the component.

For a functional component, the prop can be referred to by using props.name, where name is the name given in the parent (like first, second in *Figure 14.6*).

For a `Class` component, the prop is referred to using `this.props.name` (as `Class` component has different instances).

Props are used to pass any type of values from a parent component into a child component. This is ideal when we do not want any local data or logic at the child component level, making it a presentation only component. All logic and data can be managed by the parent, and whatever data is required for rendering content can be passed onto the child component as props.

The complete code looks like below for the class component making use of its props:

```
class ClassChild extends React.Component {
 render() {
return <h1>Hello World! I am the 1st child, my name is
{this.props.first}</h1>;
 }
}
```

If the value for the prop is not sent, the default value can be set up using defaultProps as shown:

```
// The default values for props can be setup as below
where component name is ClassChild
ClassChild.defaultProps = {
 first: "JavaScript"
};
```

To set up type checking on this props, prop-types are used. `React.PropTypes` is available in a different package since React v15.5, so it can be installed using `npm` install:

```
npm install --save prop-types
```

PropTypes

`PropTypes` can be used to associate some specific type to the props. It can also be used to add custom validations to the props.

A basic example of applying type validation:

```
import PropTypes from 'prop-types';
```

```
class MyIntro extends React.Component {
 render() {
  return (
<h1>Hello, {this.props.name}</h1>
  );
 }
}

MyIntro.propTypes = {
 name: PropTypes.string
};
```

If you now try to use Greeting component by passing a numeric value for name prop, you will get the following warning:

```
Warning: Failed prop type: Invalid prop 'name' of type
'number' supplied to 'MyIntro', expected 'string'.
```

The other types, validations which can be associated using `propTypes` are as follows:

Usage

Standard data types
`propArray: PropTypes.array,`
`propBool: PropTypes.bool,`
`propFunc: PropTypes.func,`
`propNumber: PropTypes.number,`
`propObject: PropTypes.object,`
`propString: PropTypes.string,`
`propSymbol: PropTypes.symbol`
The preceding list is optional. You can make it a required prop by adding `isRequired` to make sure a warning is shown if the prop value isn't provided.

A React element `propElement: PropTypes.element`
If prop is an instance of a class `propMessage: PropTypes.instanceOf(MyClass)`
If your prop is limited to a specific list of values, like an enum: `propEnum: PropTypes.oneOf(['First', 'Second', 'Third'])`

Your prop could be one of many types
`propUnion: PropTypes.oneOfType([`
`PropTypes.string,`
`PropTypes.number,`
`PropTypes.instanceOf(MyClass)`
`])`

Your prop could be an array of a certain type, an object with property values of a certain type. An object taking on a particular shape with given keys and value types.
`propArrayOf: PropTypes.arrayOf(PropTypes.number),`
`propObjectOf: PropTypes.objectOf(PropTypes.number),`
`propObjectWithShape: PropTypes.shape({`
`color: PropTypes.string,`
`fontSize: PropTypes.number`
`})`

Custom validation: which will check for any logic and return an Error object. The same can be applied in `arrayOf` and `objectOf` where the validation function to applied to each element.
`propCustom: function(props, propName, componentName) {`
`if (!props[propName]) {`
`return new Error("missing new country");`
`}`
`if (props[propName] === "India") {`
`return new Error(`
`"Invalid prop `" +`

```
        propName +
        "' supplied to" +
        " '" +
        componentName +
        "'. Validation failed."
      );
    }
  }
```

Table 14.1: *Prop-type validation with usage*

Passing method references between components using Props

If a method needs to be used by a child component, it can also be passed as a prop:

```
<Person
click={this.switchNameHandler}>My Hobbies: Reading</
Person>
```

In the above example, the `switchNameHandler` is being passed as a prop to the child component `Person`.

If the `handler` method expects parameters, this can be handled in two ways to ensure the this is bound to the method call along with the parameter being passed:

```
<Person
    click={this.switchNameHandler.bind(this, 'React!')}
    >My Hobbies: Reading</Person>
```

This can also be achieved using the arrow function being assigned to the handler:

```
<Person
click={() => this.switchNameHandler('JavaScript!!')}
>My Hobbies: Reading</Person>
```

Whatever logic is within the `switchNameHandler` can be invoked from the child component using the method reference which is named as **click**.

State, lifecycle and virtual DOM

We saw that props comprise of data which is passed to a component from its parent. What if the component needs to handle data of its own? The state is used to handle that!

State

This can be data required for any computations, data related to user interactions or any data related to the component behavior. State is an integral part of the component, and it comprises of data on which changes need to be tracked. The behavior of React component is such that it's render tied to the change in state. Whenever the state undergoes a change (which is not mutating), the React component and all its child components automatically get re-rendered.

The state must be initialized within the constructor function where the value of the property associated with the state is set to the initial value when the instance is created. Later in the component life cycle, the state should be manipulated using the function setState(). The state of the parent component is usually passed as a prop of the child component if the child component needs the value.

The Component state can be modified over time in response to user actions, network responses, and anything. For any modification, the component and its children are re-rendered to show the change of state.

Before React v16.8, we had to implement class-based components by extending `React.Component` to get state features, and functional components were used for presentation only, without any state. With hooks, functional components are equally powerful and can handle state. (We will explore this in detail in the hooks section. Now we will focus on class components and how they use state).

In order to use state in class components, there are five main steps as follows:

1. Identify the state structure which should comprise of the data related to the component. which may change during the component's lifecycle and impact the component behavior.
2. With structure finalized, initialize the state object in the `class` constructor. This is the only place where the state is directly assigned a value as shown:

```
 constructor(props) {
  super(props);
  this.state = {bookName: "Thinking in JavaScript",
   publicationYear: 2014 };
  }
```

3. The value of the state can be accessed using `this.state` and be used wherever needed. Since it is the state of a class component, hence the use of `this` keyword as follows:

```
render() {
  return (
<div>
```

```
<h2>Book:{this.state.bookName}, Published in :
{this.state.publicationYear}</h2>
</div>
  );
 }
```

4. Next, identify the events and interactions on which state should be updated for the component to re-render. In the current example, included a button, on click of which the state will have to change:

```
<button onClick={this.handleChange}>Click to
change!</button>
```

5. Always update the state using the `setState` method as shown. Never ever directly assign the new value:

```
handleChange(){
 if(this.state.bookName=="Thinking in JavaScript"){
  this.setState({
    bookName: "JavaScript: The Good Parts",
    publicationYear:2008
   });
  }else{
   this.setState({
    bookName: "Thinking in JavaScript",
    publicationYear: 2014
   });
  }
 }
```

In the above example, the state contains two parts: `{bookName, publicationYear}`. The value toggles between two sets of values, on the user action: the button click. In the click handler, `this.setState` method is used to change the value of the state.

Manipulating state

The state should not be mutated/changed directly by assignment as React will not recognize the change in state and the changes will not be rendered. The state should be updated using the `setState` method only as this ensures immutability.

If the state is an array, we can make use of slice or spread operator to ensure a new copy of the array is created and make changes to that. This avoids any mutable changes.

For objects also, spread operator can be used to make a copy and make changes:

```
deletePersonHandler = (personIndex) => {
  // const persons = this.state.persons.slice();//makes
  a new copy of persons
  const persons = [...this.state.persons];//makes a new
  copy of persons
  persons.splice(personIndex, 1);
  this.setState({persons: persons});
 }
```

Asynchronous setState function

`setState` is asynchronous. It does not update the state immediately in the line of execution:

```
setState({ name: "New Name" });
console.log(this.state.name); // this will not log the
value
```

Once `setState` is called, don't interact with the state until the next render, where the updated state will be available.

If we need to get the value of state and perform some action after setting state, we can use the second argument, which is a callback:

```
setState(
 { name: "New Name" },
 () => console.log(this.state)
);
```

Instead of using a callback, it is recommended to use the lifecycle methods, `shouldComponentUpdate` or `componentDidUpdate`.

Next, we will go through the different lifecycle methods and when they should be used.

Lifecycle methods

Lifecycle methods discussed in this section are specific to class components only. Functional components can handle lifecycle related events using hooks which we will discuss in the *Hooks* section.

Every React component has a life story in which it goes through some phases.

A component can be rendered using render method of another component or directly inside `ReactDOM.render` method.

On render, React instantiates an element and gives it a set of props (passed) that can be accessed with `this.props`.

Lifecycle methods are the various methods which are invoked at different

phases of the lifecycle of a component.

The React Class component which extends `React.Component` goes through the following phases:

- Mounting
- Updating
- Unmounting

Mounting

Mounting refers to the loading of components on the DOM. This phase contains a set of methods which get invoked when the component is getting initialized, and loaded on to the DOM. The following figure shows the different methods which are invoked in the process of Mounting a component.

Figure 14.9: *Mounting Phase: Lifecycle methods*

Let's look into the methods in detail:

- `constructor`: The constructor of any `class` is the very first method to get called whenever a class component is created. This constructor execution phase is called a separate initialization phase as it is used to initialize the variables and component state. This method is called only once in the entire lifecycle of a component.
- `static getDerivedStateFromProps`: As the name of this lifecycle method suggests, it is used for getting the derived state from props. This lifecycle method is useful when the local state is dependent on the value of props, such that whenever the props are changed, the local state should also be kept in sync. This method gets called after the constructor and is expected to return an object which is used to update the local state of the component. If null is returned, it implies that nothing is to be changed in the local state. The method `getDerivedStateFromProps` is static; hence it has no access to the `this` keyword. This method is a brand new addition in React v16.3+ and is a safer replacement of `componentWillReceiveProps` (older version). It should be a pure function, and nothing should be written, which creates side effects or unwanted changes.
- `render`: This is the most important method in a React component which should be mandatorily implemented. As a part of the render method, the actual element which finally gets mounted on to the browser DOM is prepared. This method is pure, which should behave consistently and return the same output every time, the same input is provided. This method also should not result in any unwanted

changes like updating state, causing side effect.

- `componentDidMount`: This lifecycle method is called after the first time the component has been rendered. This method is considered to be the best place for the API calls to fetch any external data. All the interaction directly with the browser DOM and integrate with third-party libraries like highcharts or D3 should be done here. For example, this method is also the best place to draw graphs and charts depending on data.

Updating

Once the component is rendered, there can be updates in the DOM due to user interactions or dependent component changes resulting in a change in component state or change in incoming props which triggers the **Updating Phase**. The update related lifecycle methods get invoked:

Figure 14.10: *Updating phase – Lifecycle methods*

- `static getDerivedStateFromProps`: This method behaves same as defined above in the mounting phase.
- `shouldComponentUpdate`: This function enables React to make a decision to update. This can be handled by returning true or false if we need more control on the update based on specific scenarios. This is useful when we would like to re-render the component only when the props status change:

  It is called with `nextProps` and `nextState` as the arguments:

```
shouldComponentUpdate(nextProps, nextState){ //
return a boolean value
 return true;
}
```

  By default, the `shouldComponentUpdate` returns true. If we don't want to re-render the component, then return false.
- `render`: In this lifecycle method, the updated component will be rendered to the screen with all the new data (or changes), only if `shouldComponentUpdate` returns true. If `shouldComponentUpdate` returns false, the render method is not called.
- `getSnapshotBeforeUpdate`: This lifecycle method is called after the render created the React element and before the DOM is actually updated with the differences from the virtual DOM to reflect on the actual DOM. This phase is known as the **pre-commit phase**. At this point, you can refer to both the previous and latest props and state values in this method. If the method `getSnapshotBeforeUpdate` returns a value, it is available for the `componentDidUpdate`

method as the third parameter. This is useful for scenarios where the UI has to be updated to make it synced between the before and after render status. This method is useful if you want to maintain sync between the status of the previous rendered DOM with the latest updated DOM. For example, scroll position, audio/video, text-selection, cursor position, tool-tip position, and so on.

- `componentDidUpdate`: This final lifecycle method in the updating phase gets called in any case (even if there is no change in the final output). It will get invoked as a last step of the update process, which was initiated and when the newly updated component has been updated or not updated in the DOM. This method is used to re-trigger the third-party libraries used, and to make sure these libraries also update and reload themselves:

```
componentDidUpdate(prevProps, prevState) {
  console.log('Component DID UPDATE!')
 }
```

Unmounting

In this phase, the component is not needed on the UI, and the component will get unmounted or removed from the DOM. The method related to unmounting include:

Figure 14.11: *Unmounting phase – Lifecycle methods*

`componentWillUnmount`: The component story ends with this method right before the component gets unmounted/removed from the DOM:

- In this method, we do all the cleanups related to the component.
- For example, on logout, the user details and all the auth tokens can be cleared before unmounting the main component:

```
componentWillUnmount() {
  this.resetLocalStorage();
  this.clearSession();
}
```

Now that we saw the different phases and methods a React component goes through, we will delve into another important concept behind how React works.

Virtual DOM

Virtual DOM is an in-memory JavaScript object representation of the DOM that we see in the web browser. The concept of Virtual DOM improves the performance of a React application.

Let's understand how this works:

- When we render a JSX element, every single virtual DOM object gets updated as virtual DOM can be updated very quickly.
- Once the virtual DOM has updated, then React compares the virtual DOM with a virtual DOM snapshot that was taken a right before the update.
- By comparing the new virtual DOM with a pre-update version, React figures out exactly which virtual DOM objects have changed. This process is called diffing.
- Once React knows the virtual DOM object was changed, it updates only those objects on the real DOM. Changes on the real DOM cause the screen to change.

The following figure shows the steps diagrammatically:

Figure 14.12: *How virtual DOM works?*

Styling components

CSS styles can be applied to React components in two main ways.

Styling using external CSS

This is the most common approach wherein we define external CSS files with the CSS code which need to be applied to our components. The CSS styles defined in this way have a global scope and can be used in any element in any component in the application by importing the file directly. Any component which wants to apply the style should import the CSS file explicitly.

Styling with JSX makes use of `className` instead of class to assign CSS classes to HTML elements to avoid collisions with JS classes.

The `style` class can be applied to the specific component using the `className` property:

```
<div className="App">
<h1>Hi, I'm a React App</h1>
<p>This is really working!</p>
</div>
```

Inline styling

This is defined in the specific component and applied to elements in that component only. Styles can be defined as a variable with CSS property names in CamelCase to be used in JavaScript. The styles are applied to the element using the style JSX property. The CSS properties when used as

inline style are written in lower CamelCase wherever the name is hyphen separated. For example, `background-color` is used as backgroundColor:

```
const style= { backgroundColor : 'blue',
 font : 'inherit',
 border : '1px solid red',
 padding : '8px'
}
const person = ( props ) => {
  return (
<div style={style}>
<p onClick={props.click}>I am {props.name} and I am
{props.age} years old!</p>
<p>{props.children}</p>
<input type="text" onChange={props.changed}
value={props.name} />
</div>
  )
};
export default person;
```

Inline styling has certain limitations that we cannot assign inline styles for pseudo classes like hover and media queries. Instead we need to add in global styles only which forces to be applied to the entire application.

There are some styling packages available which can be installed to handle the limitations. Styled-components, radium are two such packages which you can explore further, but this is not in the scope of this book.

Rendering lists and conditionals

We have seen how React is just JavaScript with some HTML like looking JSX and CSS styling embedded into it to build UI views. In this section, you will see how you can handle the conditional rendering of content and rendering list of items in React.

Lists

Lists are an integral part of the content, and we have seen how we can render different lists in HTML. We can use the same `ul`, `ol` and `li` tags to render a list in JSX but we will feed the data into the lists from JavaScript. Apart from the standard tags, any UI component can be repeated to render in a list format.

In order to repeat the content, depending on whether the content is in the Array format or an object, JavaScript methods like a map can be used to

render lists.

The map function can be used to traverse and return the JSX for each element as follows:

```
state = {
 courses:[
 {name: "React", prereq: "JavaScript,CSS,HTML" },
 {name: "JavaScript", prereq: "None" },
 {name: "HTML", prereq: "None" },
 {name: "CSS", prereq: "None" }
]};

render() {
 return (
<div>
<h2>Hello!</h2>
{ this.state.courses.map((course, index)=>{
       return <Course key={index} name={course.name}
       prereq={course.prereq}
       click={(e)=>this.nameChangeHandler(e,index)}/>
     })}
</div>
 );
}
```

The key prop is an important property we should add whenever rendering lists of data which helps React re-render the data in case of changes to the list. This key should hold a unique identifier for the list data.

If we don't add a key property, React gives a warning asking us to do so.

The following code makes use of a `Course.js` child component to be rendered as a list in `CourseList` parent using map function:

Figure 14.13: *The code to render a course list*

Figure 14.14: *The course list output*

Conditional rendering

Conditional rendering is as simple as using the conditional statements of JavaScript—`if…else`, ternary operator.

Now in the same example, by making a change to include a flag for conditionally rendering the text box.

```
changeFlag: true or false.

//index.js
class CourseList extends React.Component {
```

```
state = {
 courses:[
 {name: "React", prereq: "JavaScript,CSS,HTML",
 changeFlag :true},
 {name: "JavaScript", prereq: "None", changeFlag
 :false},
 {name: "HTML", prereq: "None", changeFlag :true},
 {name: "CSS", prereq: "None", changeFlag :false}
]};

nameChangeHandler(event,index){
 let courses =[…this.state.courses];
 courses[index].name= event.target.value;
 console.log(courses);
 this.setState(courses);
}
render() {
  return (
<div>
<h2>Hello!</h2>
   { this.state.courses.map((course, index)=>{
       return <Course key={index} name={course.name}
       prereq={course.prereq}
       changeFlag={course.changeFlag}
       click={(e)=>this.nameChangeHandler(e,index)}/>
   })}
</div>
  );
 }
};
```

The `changeFlag` can be used to conditionally render the input text box as follows:

```
const course = ( props ) => {
  return (
<div className="course">
<h2>Course: {props.name}</h2>
<p>Prerequisites: {props.prereq}</p>
       {props.changeFlag?<input type="text"
       onChange={props.click} value={props.name} /
       >:null}
</div>
  )
};
```

The output now looks like as follows:

Figure 14.15: *The course list output with conditionally rendering input text box*

Forms

Another important content type which we have earlier explored in our HTML chapter is forms. The same form related tags can be used in JSX for React forms. How the React forms handle the data and changes to the data categorizes forms into two types: controlled and uncontrolled forms.

Uncontrolled forms

An uncontrolled form is one that maintains its own state internally. In order to get access to the DOM elements and find the current value, you can make use of a `ref`.

Refs: Refs represent a way to be able to access the DOM nodes or React elements created in the render method. Refs are created using `React.createRef()` and can be attached to React elements using the ref attribute.

Uncontrolled form inputs are like traditional HTML form inputs, and you need to pull the value and get it from the field ref when you need it.

To write an uncontrolled component, you do not associate an event handler for every state update; instead, you can use a ref to get form values from the DOM.

Uncontrolled example:

```
class NameForm extends React.Component {
 constructor(props) {
  super(props);
  this.handleSubmit = this.handleSubmit.bind(this);
  this.input = React.createRef();
 }

 handleSubmit(event) {
  alert('A name was submitted: ' +
  this.input.current.value);
  event.preventDefault();
 }

 render() {
  return (
<form onSubmit={this.handleSubmit}>
<label>
       Name:
<input type="text" ref={this.input} />
```

```
</label>
<input type="submit" value="Submit" />
</form>
  );
 }
}
```

Controlled forms

Unlike uncontrolled forms which let the HTML elements maintain their own state, the controlled forms maintain React state as the “single state of truth”.

The React component that renders a form will also keep track of what happens with the form data, what value is entered by the user.

A controlled component is one that has its current value tied to React state, gets initial value from the state, and any changes to the value are notified and handled to change the state.

Let's look at an example for controlled components:

```
class NameForm extends React.Component {
 constructor(props) {
  super(props);
  this.state = {value: ''};

  this.handleChange = this.handleChange.bind(this);
  this.handleSubmit = this.handleSubmit.bind(this);
 }

 handleChange(event) {
  this.setState({value: event.target.value});
 }

 handleSubmit(event) {
  alert('A name was submitted: ' + this.state.value);
  event.preventDefault();
 }
 render() {
  return (
<form onSubmit={this.handleSubmit}>
<label>
      Name:
<input type="text" value={this.state.value}
onChange={this.handleChange} />
</label>
<input type="submit" value="Submit" />
```

```
</form>
    );
  }
}
```

Differences between controlled and uncontrolled forms:

- In a controlled component, form data is handled by a React component, whereas in uncontrolled components, the form data is handled by the DOM itself.
- Uncontrolled forms are used for simpler forms with few inputs, whereas controlled forms are preferred for larger forms where all input data changes need to be tracked and handled.

Composition

The concept of composition in React is simply how different components combine to make up bigger components, which finally make up the UI. The components can be reused wherever needed as part of another component.

A Base component can be created first, and any specialized version can include the base component and any additional logic or components to compose bigger and more specialized components.

Hooks

We heard the mention of hooks multiple times in the earlier sections. Do you remember? What do hooks provide?

Hooks add the power of the state and lifecycle methods to functional components so that functional components can now be used to define smart components rather than the presentation only components as before React v16.8 (hooks).

Hooks also enable to reuse the logic associated with them across multiple functional components. We cannot do this with class components as its methods and properties belong to a specific instance. In this section, we will explore the important hooks and what functionality they bring on the table.

The useState hook

The `useState` hook gives the ability to functional components to be able to use local state. This function hook takes the initial value of state as an argument and returns a two-element array containing the current state value, and a function to update that value.

`useState` takes the initial value as the input which is an empty `array[]` in the following example and returns the current state and the function to update the state (equivalent to the `setState` function in class components):

```
const [users, setUsers] = useState([]);
```

Now, we can get the current state using the first value returned (`users`) and use the second value returned (`setUsers`) if we need to update the state.

All of this can be achieved without the keyword this being attached to it, as all this pertains to a function and not a class. This gives us a similar behaviour in functional components as class components with lighter code.

The useEffect hook

Earlier you learnt that in class components, we handle any side-effects like API requests, manual DOM mutations, and logging using the lifecycle methods `componentDidMount` and `componentDidUpdate`.

The `useEffect` hook handles the effects in the same way for functional components providing similar capability with better flexibility. This hook contains the logic to be executed after the first render and every consecutive render, like `componentDidMount` and `componentDidUpdate`.

The `useEffect` takes one mandatory parameter, which is the function that gets executed after render. The default behaviour of this function is that it gets executed after each time the render method gets executed.

This can be optimized by making use of the second optional parameter, the dependency that controls whether or not the function needs to be executed.

If passed as an [] (empty array), which implies no dependency, it gets executed only once. If passed as some value like [`users`], it gets executed only when there is a change in the dependent value, being passed as the second parameter.

In the following example, we will be using Axios, a client to make HTTP calls to external servers to fetch and post data. You will have to install `axios` as a dependency for the project using the following command:

```
npm install --save axios
```

For testing, you can create a test database in firebase.google.com and enter test data for users which can be accessed using `axios`.

For the user fetch example using `useState` and `useEffect` hooks, only to get the user data in `useEffect` as follows:

```
const Users = props =>{
const [users, setUsers] = useState([]);// set state
useEffect(() => { // will get executed after render
console.log("useEffect called");
axios.get('https://hooks-demo-a6d15.firebaseio.com/
users.json')
.then(res => {
const users=[]
for (const key in res.data){
users.push({ id: key,
name :res.data[key].name,
age: res.data[key].age,
interests: res.data[key].interests});
}
setUsers( users );
})
.catch(err => {
console.log(err);
});
},[]);// the second parameter is[] so will be called
only after first render
```

This is the optimal approach if we want to fetch the data again based on some dependent value like when the search criteria or the dependent data changes.

How can hooks be reused?

The hook-related logic can be separated into a custom hook (name prefix used for example, `useUsers.js`). It can be imported and invoked like a function in any other functional components that need to use the same logic:

```
import { useState, useEffect } from 'react';
import axios from 'axios';
const useUsers = props =>{

//useState hook to initialize state
const [users, setUsers] = useState([]);

//useEffect hook to define the functionality to get the
data
useEffect(() => {
console.log("useEffect called");
axios.get('https://hooks-demo-a6d15.firebaseio.com/
users.json')
```

```
.then(res => {
const users=[]
for (const key in res.data){
users.push({ id: key,
name :res.data[key].name,
age: res.data[key].age,
interests: res.data[key].interests});
}
setUsers( users );
})
.catch(err => {
console.log(err);
});
},[]);
return users;
};

//finally export this entire functionality as a custom
hook named useUsers
export default useUsers;
```

Now, it can be imported and used in a simple way, as demonstrated as follows in any of the functional components:

```
import useUsers from './useUsers';
const users = useUsers();
```

This could prevent code duplication and make our functional components lighter and easier to understand.

These are the main hooks which empower the functional component. You can explore the hooks further in React, but these two hooks are the basic hooks which make functional components equally capable as class components.

Conclusion

In this chapter, you were introduced to all the basic concepts of React, a powerful UI Library. You are now fully empowered to make highly interactive and performant applications using React in the proper way. In the next chapter, we will use this newly acquired React knowledge and build a complete application.

Questions

1. Functional components

A. Cannot handle state

B. Cannot handle lifecycle events

C. Can handle state and lifecycle events using hooks

D. Are always stateless

Answer: Option C

Functional components can handle state and lifecycle events using the useState and useEffect hooks.

2. Which of these is correct syntax for passing method reference in props?

A. `click={() => this.switchNameHandler('React!!')}`

B. `click={this.switchNameHandler.bind(this, 'React!')}`

C. `click={this.switchNameHandler}`

D. All of the above

Answer: OptionD

All the given statements are correct. A and B are used for passing parameters along with the method and C is used if there are no parameters.

3. The ____ prop is an important property to be added when rendering lists of data.

A. Name

B. Id

C. key

D. Src

Answer: OptionC

The key attribute should be used whenever rendering lists by providing a unique value for each entry of the list which enables React to re-render the list for any changes and uniquely identify which list element has changed.

4. Match the following properties with the lifecycle methods

i. This is used to set up the initial values of variables and component state.

ii. This is used for API calls to fetch data.

iii. This is used when the state is dependent on props, and hence whenever the props are changed, the state has to be kept in sync.

iv. This is useful when we would like to re-render the component

only when the props status changes.

1. `getDerivedStateFromProps()`
2. `constructor()`
3. `shouldComponentUpdate()`
4. `componentDidMount()`

A. i-1, ii-2, iii-3, iv-4

B. i-2, ii-4, iii-1, iv-3

C. i-2, ii-3, iii-4, iv-1

D. i-4, ii-1, iii-2, iv-3

Answer: OptionB

The constructor is used to set up the initial values of variables and component state. The `componentDidMount()` lifecycle method is for API calls to fetch data. The `getDerivedStateFromProps()` method is used when the state is dependent on props, and hence whenever the props are changed, the state has to be kept in sync. The should `ComponentUpdate()` is useful when we would like to re-render the component only when the props status changes and we want to check and decide to go ahead with the update or not.

5. By comparing the new virtual DOM with a pre-update version, React figures out exactly which virtual DOM objects have changed. This process is called _____.

A. Referencing

B. Differentiation

C. Diffing

D. None of the above.

Answer: Option C

Diffing is the process of comparing the new virtual DOM with the pre-update version using which React decides if there is any real update to be done to the Real DOM.

Weather

Search

Current weather in Kukatpalli

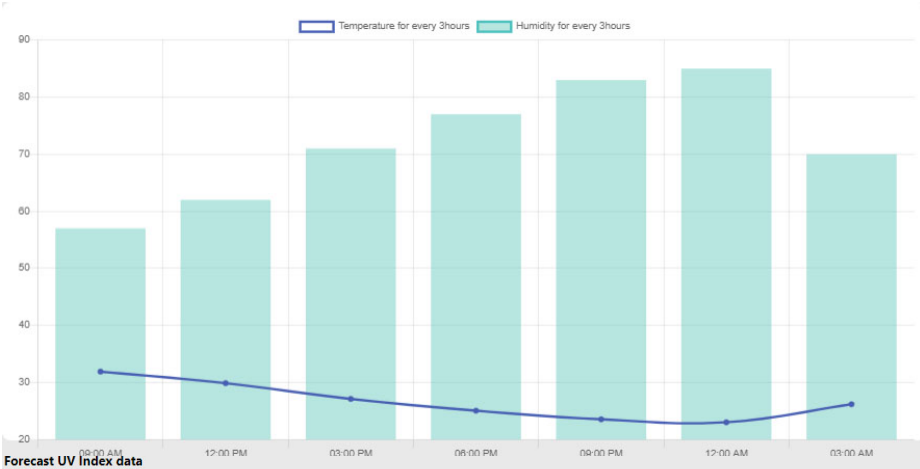
Current Temperature	Current Pressure	Current Humidity	Current Wind
29.29 °C	1010 hpa	70 %	6.2 m/sec

Summary weather report

29.29^{0}C

haze

Wind	6.2 m/sec
Cloudiness	haze
Pressure	1010 hpa
Humidity	70 %
Sunrise	05:48 AM
Sunset	06:54 PM
Geo coords	17.48,78.39



Forecast UV Index data

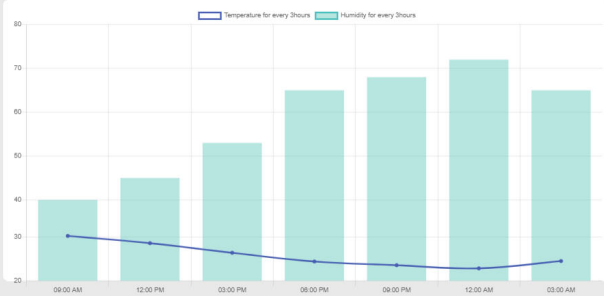
Day	2019-07-12	2019-07-13	2019-07-14	2019-07-15	2019-07-16	2019-07-17	2019-07-18	2019-07-19
Uv value	14.69	13.84	13.44	13.78	14.17	13.66	13.54	14.45

Weather

Bangalore Search

Current weather in Bangalore

Current Temperature	Current Pressure	Current Humidity	Current Wind
28.52 °C	1013 hpa	58 %	7.2 m/sec



Summary weather report

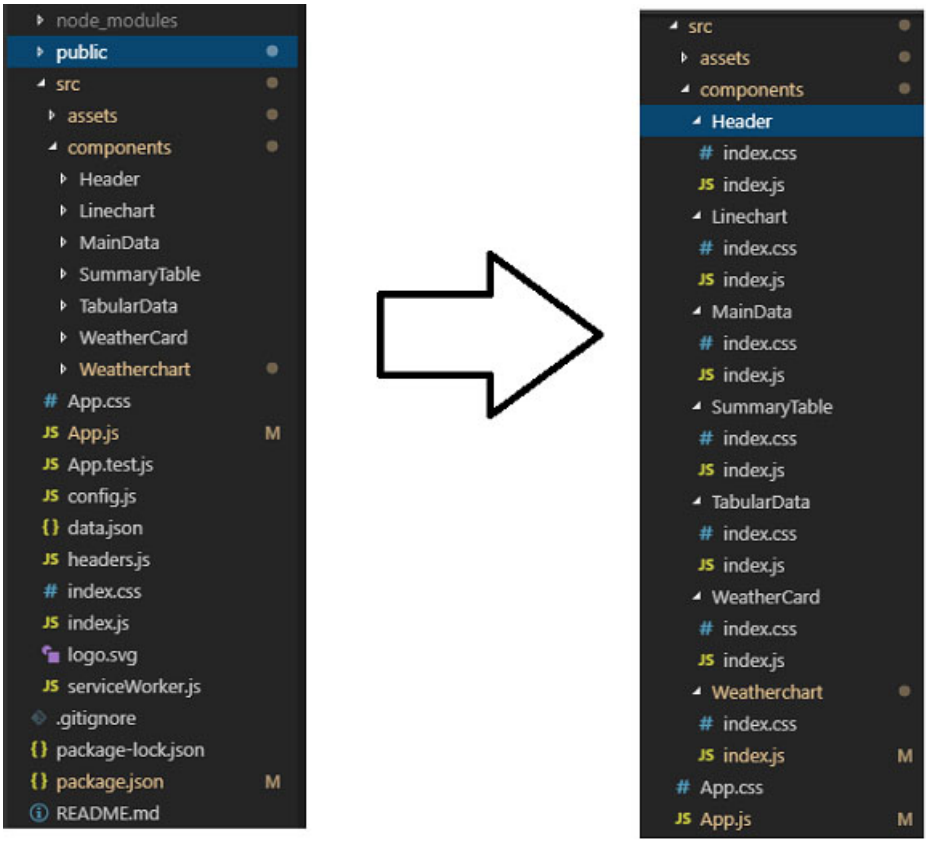
28.52^{0}C

scattered clouds

Wind	7.2 m/sec
Cloudiness	scattered clouds
Pressure	1013 hpa
Humidity	58 %
Sunrise	05:59 AM
Sunset	06:50 PM
Geo coords	12.97,77.59

Forecast UV Index data

Day	2019-07-12	2019-07-13	2019-07-14	2019-07-15	2019-07-16	2019-07-17	2019-07-18	2019-07-19
Uv value	13.56	14.13	15.26	14.65	14.71	14.88	15.15	14.81



CHAPTER 15

Building an Application with React

"Success is no accident. It is hard work, perseverance, learning, studying, sacrifice and most of all, love of what you are doing or learning to do."

~ Pele

In the previous chapter, we learned about React and its basic concepts of state, props, JSX, and hooks. After having learned all the concepts, it's now time to apply what you have learned and see them in action. In this chapter, we will go through the development of a simple React application step by step right from understanding the requirements, designing our application and finally, building the application. We will learn to apply the concepts to build a fully functional application.

Structure

- Thinking in React to approach the UI development
- Requirements and application design
- Prerequisites and getting started setup
- Building the React application

Objective

At the end of this chapter, you will know about the important aspects of building a fully functional React application.

Thinking in React to Approach the UI Development

Before we get into building the React application, let's first understand few basic steps which you should keep in mind before approaching any application development:

- **Prepare your mockup:** The first and most important aspect when we have to build an application is to have a complete clarity of what the end product should look like. UI screens should be designed, discussed, and finalized before starting any real development. This may not be possible in some situations if you are still in discussions with the client who wants you to start with the development in parallel with the discussions. But it is always better to have the user experience clearly defined and agreed upon in the form of UI mockups.
- **Plan your UI components:** Your React application is made up of multiple React components put together instead of one complex and huge component. Once you know how your screens will look like, you should plan out how you will split up each screen into different components. This will also give a visibility of reusable components which can be used across multiple screens like a common header and footer component. It is a good idea to have each component cater to one aspect or one problem of the application. This is referred to as a **Single Responsibility Principle.** Along with the component definition, also look for any need of component composition where the same base component can be used to build a more specific component. At the end of this activity, you should have a component hierarchy which lays out all the planned components and how they relate to each other.
- **Identify your data:** The look and feel of the application which we

have defined so far has no meaning without proper data to show and interact. The next important activity is to identify the data requirements of your application. When defining the data, some of the following questions are to be answered:

- What are the different data elements?
- What is the structure of the data?
- From where will the data be fetched?
- Will the data undergo any changes in the application? If yes, what part is static and what will change?
- What changes are to be tied to the state of the application to trigger re-render?
- Does the change need to be persisted?
- How are the changes propagated back to the server?
- What will be the structure of the state of the application?

- **Link up data with UI Components:** With the picture of UI at one side and the data at the other side, now you should be able to link both aspects and have the complete application design. This process will be ongoing as you start your application development and some questions which will be answered include the following:
  - Which components will be presentations only and will receive the data as props?
  - Which UI components will hold the state of their own and need to subscribe to the changes of state?
  - How will the state be changed and handled in the component?

At the end of this thinking process, you will come out with a clear idea of what you want to build, what is the data and state you are dealing with, and how they will work together when your application comes into action. There may be changes to the initial plan as you go ahead with the real development as things may change or data may behave differently, but having a clearly thought out execution will minimize the impact of any such changes and they can be easily incorporated.

Now, we will apply this thought process to our sample application and see how we bring it into action.

Requirements and Application Design

In this chapter, we will build a weather application which will show the weather data in different formats using styling and third-party libraries. The data will be fetched from an open source weather API and used to render it on the screen.

By default, the application will render the data of the user location. The user can also search for other locations and the application will render the weather details of the searched location.

Let's look at the following steps to proceed with the application design:

1. **Prepare your mockup:**

   Starting with the first step to have a mockup, this application comprises the following screen which will display the weather data in graphs, cards, and tables.

   (In case you are wondering, Kukatpalli is my current location in Hyderabad, where I am working on this application:

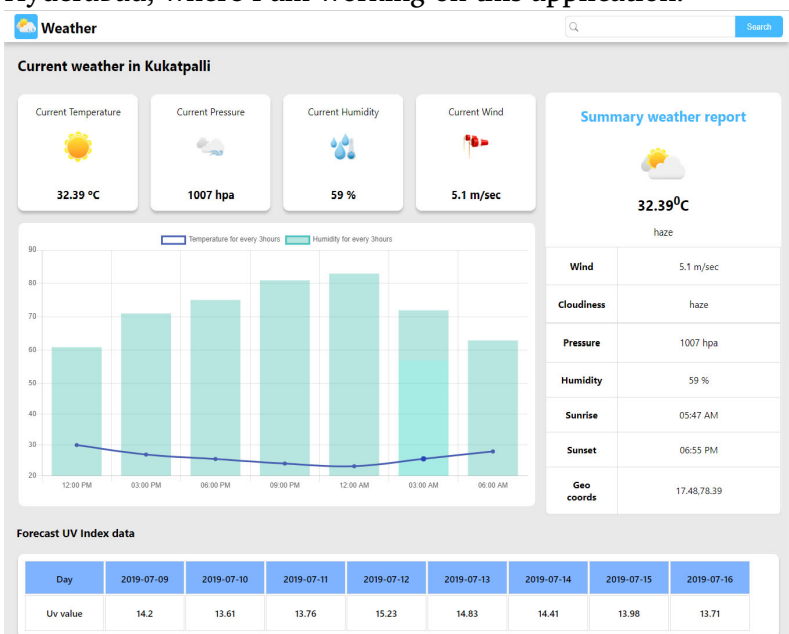


Figure 15.1: *The weather application*

   The application will give the preceding dashboard with the current weather and some forecast statistics of the searched location (by default, it will show the user's current location).

Before you start building any application, we will look at the other two aspects which should be planned:

1. **Plan your UI Components:**

   A React application UI is made up of multiple React components put together.

   Whenever we approach a React application, **User Interface (UI)**, we divide the UI into the different components which will be defined separately and which will be put together to make up the complete

UI.

In the current UI, we can plan for the following components:

- `App`: The parent container component of the application
- `Header`: Includes the **`Icon`** with the **`Search`** bar
- `MainData`: Container component to get the weather details and pass on to the other components
- `WeatherCard`: Component for the set of weather cards with the current weather data
- `WeatherChart`: The bar and line chart showing the three-hourly temperature and humidity data
- `SummaryTable`: The summary weather details shown in a tabular format
- `TabularData`: The forecast UV Index data

Some of these components may not come up in the initial plan; for example, the `MainData` component is a parent container which passes the data to the three child components and you may not plan for it initially but may come up later when you see a need of sharing data between components to have a common container.

The component hierarchy is as follows:

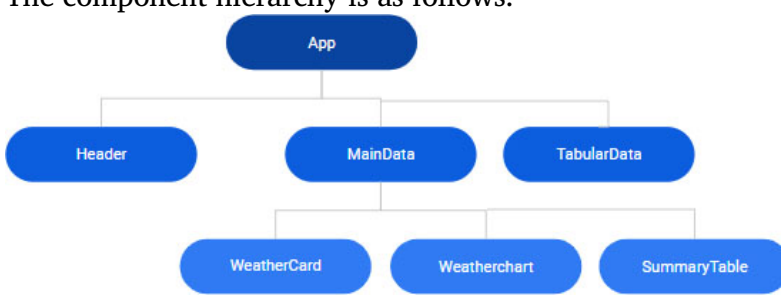


Figure 15.2

9. **Identify your data:**

The current application deals with the location and the weather data of the selected location. There are various parts of the weather application data which include the following:

- The search location and its coordinates
- The current temperature, pressure, humidity, wind, sunrise, sunset, and cloudiness
- The temperature and humidity for every three hours to be used for plotting the charts
- The forecast UV index data

An open source API, **https://openweathermap.org/**, will be used to fetch the weather details using a third-party library called **axios**. In order to work with the API, you will have to sign up and get an API Key which will be used while accessing the data using this API.

Another open source **https://www.bingmapsportal.com/** will be used to fetch the coordinates for a given location. You will have to sign up into this and get the API key to be used.

Both the API keys can be maintained separately in a `config.js` file and imported into the components when needed to make an API call.

14. **Link up data with UI components:**

    This being a simple app, we can look into the linking up of data with the UI components as we get into the details of the application building.

    On a high level, we have the following considerations for the components:

    - `App`: Stateful component, state contains the search location and its coordinates
    - `Header`: Presentation or stateless component, search data received and passed using props to parent app
    - `MainData`: Container component to fetch the weather details and pass on to the other components gets coordinates from `App` as props
    - `WeatherCard`: Presentation component, which gets data from `MainData` component as props
    - `WeatherChart`: Presentation component, which gets data from `MainData` component as props
    - `SummaryTable`: Presentation component, which gets data from `MainData` component as props
    - `TabularData`: Stateful component for forecast UV index data, which fetches its own data and gets coordinates from `App` as props

After having the initial design and solution approach of our application, now it's time to get started with actual development.

Prerequisites and getting started

Every React application will start with the first step to build a template app as shown in *Chapter 14, Introduction to React*. Let's follow the same steps and create our new application. Let's call our app my-weather-app and create the base app using the following statement:

```
npx create-react-app my-weather-app
```

This statement creates the my-weather-app folder with the default folder structure for our project.

We go to the project folder and will first install the dependencies for this project. We can do this upfront if we know what all we would need or can install more as and when we encounter a new dependency required during development. Here's my initial list of dependencies:

- **axios:** To make API calls
- **chart.js and react-chartjs-2:** To render the bar and line charts for weather data

Just like the preceding libraries for charts, you can find thousands and thousands of supporting libraries in NPM which can help you provide additional functionalities and depending on your application needs, you can include them as dependencies and use them in your React components. Always look for existing libraries which can be reused to provide for the functionality. If you end up with a new requirement which does not match with any existing library, you can publish your own solution on NPM for the world to use.

Getting back to the application folder, run the following command to install the listed additional dependencies:

```
npm install axios chart.js react-chartjs-2 --save
```

Building the React application

Now with the dependencies in place, let's start building the React application.

Step 1

First, let's understand our application screen and how we approach it. Let's take the UI piece by piece and build it.

Let's create a sub folder called components under the src folder and add all the new components inside it. Also, for the components, we will follow the following structure:

- A folder will be created for each component.
- An `index.js` will be created with the React code of the component, which will be picked by default by webpack at the time of bundling.
- An `index.css` stylesheet file will be created to define style classes for the specific component.

Next, let's approach each UI component one by one. The import statements

have not been included in the following listed piece of code, but imports will have to be included wherever external components are being referred and used.

Step 2

We will start the `App.js` which will have the common state defined as follows:

```
this.state = {
    latitude: “”,
    longitude: “”,
searchTerm:””,
   }
```

The state includes the following parts:

- `Latitude and longitude`: The coordinates of the location which is being searched for.
- `SearchTerm`: The term which is given to search for the location.

The `App.js` code is as follows. We create a class component here:

```
//Class component
class App extends Component {
newState={} ;// a newState variable used to update
state
activeCoords={}; // to handle the current coordinates
 constructor(props) {
  super(props);
 // the state will be initialised to empty values
this.state = {
    latitude: “”,
    longitude: “”,
searchTerm: “”,
   }
   //This Method call is used to get the coordinates of
   the current location of user. This is initialised in
   the constructor so that the application loads with
   the default weather of the current location
navigator.geolocation.getCurrentPosition(position => {
constcoords = position.coords;
this.newState = {
     latitude: coords.latitude,
     longitude: coords.longitude,
    }
this.setState(this.newState);
```

```
  });
 }
//The getCoordinates function will be used if the user
enters a different search location, to get the
corresponding coordinates
 //Function which converts given address or city name
 to latitude and longitude on change of the city name
getCoordinates=(city) => {
  if (city === "Current location") {
this.setState(this.newState);
  } else {
   let locality = city,
countryRegion = 'IN';//Region is set to India
     axios.get('http://dev.virtualearth.net/REST/v1/
     Locations?&countryRegion=${countryRegion}
     &locality=${locality}&key=${geocoding_Api_key}
 `).then((response) => {
//The response is mapped and the coordinates extracted
for the location
response.data.resourceSets.map((data) => {
    return ( data.resources.map((resourceData) => {
     if (resourceData.confidence === "High") {
this.activeCoords={
      latitude: resourceData.point.coordinates[0],
      longitude: resourceData.point.coordinates[1]
     };
     console.log(this.activeCoords);
     return this.activeCoords;
     } else {
console.log('no data');
     return null;
     }
    })
    )
    })
   })
  } }
 //Method which handle the change of input entered in
 search box and set it to state searchTerm
handleChange=(e) => {
constnewSearch= e.target.value;
this.setState({
searchTerm: newSearch
  });
```

```
this.getCoordinates(newSearch);// keep the coordinates
updated based on the city name entered
 }
//Method updates the coordinates in the state so that
the child components are rerendered for new
coordinates. This is invoked on click of search button
in Header component. It is passed as a reference from
the App component.
handleSearch=(e) => {
let coords= this.activeCoords;
this.setState({
       …this.state, //Changing the state with search
       term latest coordinates
       …coords
      }, () => console.log(this.state));
  }
 //render method, which will contain all the jsx which
 finally gets rendered on the UI. This includes all the
 child components included in App component
render() {
  if (this.state !== null) {
   return (
<div className="App">
<Header handleSearch={() =>this.handleSearch()} //
Header Component
handleChange={(e) =>{ this.handleChange(e) } />
<div className="container">
<MainDatacoords={this.state}/> {/* Main Component which
consists cards,Summary weather table,Weather Chart*/ }
</div>
<TabularDatacoords={this.state}/> {/* which consists Uv
index tabular data for 8days ahead*/}
</div>
    );
  }
  else
   return <div></div>
 }
}
export default App;
```

The render function of App renders three components, namely, `Header`, `MainData` and `TabularData`.

We will take up each one of them and explain the code for the same.

Step 3

First, we will begin with the Header component which will be created as a functional component.

The `Header/index.js` file will contain the logic to render the header of the page with the logo on the left and the search bar at the right, as follows:

```
//Functional Component of header
const Header = (props) => {
// The data being shown in the Header includes the logo
and the search bar which is an input datalist
//The handleChange and handleSearch method reference is
passed as props from parent App component
 return (
<div className=”headNavbar”>
<div className=”logo_container”> {/*div Consists logo
of App*/}
<imgclassName=”logo” src={Logo} alt=”logo”/>
<h2 className=”logo_name”>Weather</h2>
</div>
<div className=”searchBar”>
<div className=”search-container”>
<imgclassName=”search_icon” src={SearchIcon}
alt=”SearchIcon”/>
<input list=”browsers” name=”browser”
className=”search_field” onChange={props.handleChange}/
>
<datalist id=”browsers” >
<option value =”Current location”/>
<option value=”Hyderabad”/>
<option value=”Kolkata”/>
<option value=”Bangalore”/>
<option value=”Chennai”/>
<option value=”Mumbai”/>
</datalist>
</div>
<button className=”search_button” type=”submit”
onClick={props.handleSearch}>Search</button>
</div>
</div>
 )
}
export default Header;
```

The `Header` component will render the header part of the UI as shown in the following image:

Figure 15.3: *The Header component of the Weather application:*

With the header in place, we will start the next component MainData.

Step 4

The `MainData` container component includes all the data-related charts and graphs for which data will be fetched in this component using the following URL **https://api.openweathermap.org/data/2.5/weather?lat=${latitude}&lon=${longitude}&units=metric&appid=${Api_key}** and passing the location coordinates. This component receives the coordinates as props and uses them to fetch all the weather-related data. Whenever the user enters a new location and clicks on the **Search** button, the state corresponding to the coordinates is updated in the parent component (which is App) which triggers a re-render of this and the other child components and they get the new value of props. Now, the weather data is fetched for the new prop value and new data is rendered as shown in following code:

```
constMainData=(props)=>{
const {latitude, longitude} = props.coords;
//State is defined as different pieces of information
required by the component and its children

const [weatherData, setWeatherData] = useState({});
const [mainData, setmainData] = useState({});
const [windData, setwindData] = useState({});
const [sunData, setSunData] = useState({});
const [cloudData, setCloudData] = useState({});

 //The useEffect hook is used to fetch the data like a
 lifecycle hook for functional component as it will
 automatically refetch on change of the second
 parameter [latitude, longitude]

useEffect(() => {
axios.get(`https://api.openweathermap.org/data/2.5/
weather?lat=${latitude}&lon=${longitude}
&units=metric&appid=${Api_key}`).then((response) => {
//setting the different elements from the API response
setWeatherData(response.data);
setmainData(response.data.main);
setwindData(response.data.wind);
setSunData(response.data.sys);
```

```
setCloudData(response.data.weather[0]);
    })
.catch(err => {
      console.log(err);
     });
  }, [latitude, longitude]);

//If data is received, render the desired view by
returning the JSX to be rendered. In this case it
includes the child components WeatherCard, Weatherchart
and SummaryTable

if(weatherData&&
weatherData!==null){
return(
<div className=”main_container”>
<div className=”card_container_main”>
<WeatherCardweatherData={weatherData}
mainData={mainData}
windData={windData} />
<Weatherchartcoords={props.coords}/>
</div>
<SummaryTableweatherData={weatherData}
mainData={mainData}
windData={windData}
sunData={sunData}
cloudData={cloudData}
coords={props.coords}/>
</div>
  )
}else{
return(
<div></div>//empty view if no data available
  )
 }
}
export default MainData;
```

The `MainData` component renders the following:

- `WeatherCard`
- `SummaryTable`
- `Weatherchart`

The common weather data is fetched in `MainData` and passed as props to the child presentation components.

Step 5

`WeatherCard` is the first presentational component which receives the complete data from its parent `MainData` and renders a card layout.

This will render the cards for `temperature, pressure, humidity,` and `wind` which is initialized with the variable `card_data` and gets the value from props as shown in the following code:

```
constWeatherCard = (props) => {
//The card data being initialised with props values
constcard_data=[{
    'Header':'Temperature',
    'cardData':'${props.mainData.temp} \xBAC',
    'src':Temperature
  },{
    'Header':'Pressure',
    'cardData':'${props.mainData.pressure} hpa',
    'src':PressureIcon
  },{
    'Header':'Humidity',
    'cardData':'${props.mainData.humidity} %',
    'src':HumidityIcon
  },
  {
    'Header':'Wind',
    'cardData':'${props.windData.speed} m/sec',
    'src':windIcon
  }
  ]
if (props.weatherData !== null) {//to check if the
weatherData is available
 return (
<div className="card_container">
<h2 className="cards_header">Current weather in
{props.weatherData.name}
</h2>
<div className="cards">
   {card_data.map((data,i)=>{
return(
<div className="weather_cards" key={i}>
<p>Current {data.Header}</p>
<div className="icon_container">
<imgclassName="weather_icon" src={data.src}
alt="weatherIcon"/>
</div>
```

```
<h3>{data.cardData}</h3>
</div>
     )
    })}
</div>
</div>
  );
} else {
 return <div></div>
// }
}}
export default WeatherCard;
```

The `WeatherCard` component will render the four cards with the current weather details as shown in the following image:

Figure 15.4: *The WeatherCard component showing the current weather details in the Weather application*

Step 6

`SummaryTable` is the second presentational component as it shows the same data fetched in the API call by the `MainData` component. Hence, it receives the data as props as follows:

```
constSummaryTable = (props) => {
//Function which converts Unix date to local date
constdateConverter=(data)=>{
  let localDate = new Date(data*1000);
  let date= localDate.toLocaleString(undefined, {
 day: 'numeric',
 month: 'numeric',
 year: 'numeric',
 hour: '2-digit',
 minute: '2-digit',
 }).slice(10,19)
  return date;
 }
constsunriseTime=dateConverter(props.sunData.sunrise);//
Function calling for date conversion
constsunsetTime=dateConverter(props.sunData.sunset);
constnew_latitude =
parseFloat(props.coords.latitude).toFixed(2);
//Method which rounds the decimal number to fixed
2points
constnew_longitude =
```

```
parseFloat(props.coords.longitude).toFixed(2);
 //Summary weather data object
constsummary_data=[
   {
   'Header':'Wind',
   'tableData':`${props.windData.speed} m/sec`
   },{
    'Header':'Cloudiness',
    'tableData':`${props.cloudData.description}`
   },{
    'Header':'Pressure',
    'tableData':`${props.mainData.pressure} hpa`
   },{
    'Header':'Humidity',
    'tableData':`${props.mainData.humidity} %`
   },{
    'Header':'Sunrise',
    'tableData':`${sunriseTime}`
   },{
    'Header':'Sunset',
    'tableData':`${sunsetTime}`
   },{
    'Header':'Geocoords',
    'tableData':`${new_latitude},${new_longitude}`
   },
   ];
//The formatted table with the data is returned for the
view

return(
<div className="summary_table">
<h2 className="summary_header">Summary weather report</
h2>
<div className="display_Something">
<div>
<imgclassName="summaryWeatherIcon"
src={SummaryWeatherIcon} alt="weatherIcon"/>
<h2 className="temperature">{props.mainData.temp}
<sup>0</sup>C</h2>
<p>{props.cloudData.description}</p>
</div>
</div>
<table className="table_grid" >
      {summary_data.map((data,index)=>{
return(
```

```
<tbody key={index}>
<tr>
<th>{data.Header}</th>
<td className="summary_td">{data.tableData}</td>
</tr>
</tbody>)
       })
      }
</table>
</div>
  )
}
export default SummaryTable;
```

This component also does not make any changes to data but only presents the data received as props from the parent `MainData` component.

This makes use of an HTML table to display the weather summary data in the form of a table as shown in the following image:

Figure 15.5: *The SummaryTable component showing summary of weather information*

As you can see, we are building the application piece by piece.

Step 7

The next part is the `WeatherChart` which shows the three-hourly forecast data for temperature and humidity in the form of a chart needs to get data from the following API **https://api.openweathermap.org/data/2.5/forecast?lat=${latitude} &lon=${longitude} &units=metric&cnt=7&appid=${Api_key}**.

The location coordinates are passed as props which will be used to fetch the forecast data from the API as follows:

```
const styles = {
graphContainer: {
  padding: '1.2%',
  width:'94.5%',
  height: '100%',
backgroundColor: 'white',
marginLeft: '1.2%',
  borderRadius:'10px'
 }
}
// Functional component which has chartData as its
state
constWeatherChart = (props)=>{
```

```
const{ latitude, longitude } = props.coords;
const [chartData, setchartData] = useState({});

useEffect(() => {
axios.get('https://api.openweathermap.org/data/2.5/
forecast?lat=${latitude}&lon=${longitude}
&units=metric&cnt=7&appid=${Api_key}').then( (response)
=>{
setchartData(response.data);
    })
.catch(err => {
     console.log(err);
    });
},[latitude,longitude]);
//The chart data is set up in the required format and
passed into the Chart related component, Bar
//The chart component expects data in a predefined
format

if(chartData.list&&
chartData.list!==null){
varapiData={
labels:chartData.list.map((data)=>{
       let time = data.dt_txt.slice(11,16);
       let Hours = +time.substr(0, 2);
       let hour = (Hours % 12) || 12;
       hour = (hour < 10)?("0"+hour):hour;
       let ampm = Hours <12 ? " AM" : " PM";
       time = hour + time.substr(2, 3) + ampm;
 return time;
     }),
    datasets: [
      {
type:'line',
       label: 'Temperature for every 3hours',
data:chartData.list.map((data)=>{
        return data.main.temp }),
backgroundColor :'rgba(0,0,0,0)',
borderColor :'#475eb2',
pointBackgroundColor:'#475eb2'
      },
      {
type:'bar',
label:'Humidity for every 3hours',
       fill: false,
```

```
lineTension: 0.1,
backgroundColor: 'rgba(75,192,180,0.4)',
borderColor: 'rgba(75,192,192,1)',
barPercentage: 0.2,
data:chartData.list.map((data)=>{
       return data.main.humidity }),
       }
      ]
    }
  }
const options = {
   legend: {
     display: true,
      },
   };
  return (
<div className="weather_container">
<div style={styles.graphContainer}>
<Bar
className="weather_chart"
    data={apiData}
    options={options}
    />
</div>
</div>
  )
 }
export default WeatherChart;
```

Using the chart-related libraries with required parameters, the chart data is shown as follows:

Figure 15.6: *The Weather Chart component showing the three-hourly forecast data for temperature and humidity*

This puts up the complete `MainData` component.

Step 8

The next component to show the UV index data for eight days ahead in a tabular format is `TabularData`. To get the UV index data, this component needs to use a different API **http://api.openweathermap.org/data/2.5/uvi/forecast?appid=${Api_key}&lat=${latitude}&lon=${longitude}&cnt=8**.

This component also makes use of the table-related elements to render the output in a tabular format as follows:

```
constTabularData = (props) => {

const {latitude, longitude} = props.coords;
const [tabularData, setTabularData]=useState([]);

//The useEffect hook is used with logic to fetch the
data whenever the coordinates change

useEffect(()=>{
axios.get(‘http://api.openweathermap.org/data/2.5/uvi/
forecast?appid=${Api_key}&lat=${latitude}&lon=
${longitude}&cnt=8’).then( (response) =>{
setTabularData(response.data);
  })
  }, [latitude, longitude]);
if(tabularData
&&tabularData!=={}){

//The table view is returned by this functional
component with the data formatted as per requirement

return(
<div>
<h3 className=”header”>Forecast UV Index data</h3>
<div className=”table_data”>
<table >
<thead>
<tr>
<th>Day</th>
      {tabularData.map((data,i)=>{
return(
<th key={i}>{data.date_iso.slice(0,10)}</th>
      )})}
</tr>
</thead>
<tbody>
<tr>
<td>Uv value</td>
      {tabularData.map((data,index)=>{
return(
<td key={index}>{data.value}</td>
       )})}
</tr>
</tbody>
</table>
</div>
```

```
</div>
   )
  }
else{
return(
<div></div>
    )
   }
}
export default TabularData;
```

The `TabularData` component looks like the following screenshot:

Figure 15.7: *The TabularData component showing the forecast UV index for next eight days*

All these components put together build the entire weather application. You can query for a different city by entering or selecting the name and clicking on the **`Search`** button and it will render the updated weather details for the searched location as shown.

On change of location, the state is updated with the new search term and new coordinates are fetched. As all the other components are receiving the coordinates or the weather data as props, all will get re-rendered with new data.

So, if you search for Bangalore, you will see the details as shown in the following screenshot:

Figure 15.8: *Searching for Bangalore renders the weather details of Bangalore*

All the components have been given some styling by having some supporting classes defined in CSS files associated with each of them.

You can view the CSS files in the code bundle provided along with this book under `WeatherApp`.

The overall folder structure of the project and the expanded `src` folder showing the component folders looks like the following screenshot:

Figure 15.9: *Project folder structure*

Congratulations! You just completed your first full-fledged React weather application. You can check the complete project code, including the stylesheets in the code bundle under the folder `/Chapter-15&16-my-weather-app/my-weather-app/Only-React`.

Conclusion

By following the steps in this chapter, you should be able to build a weather application to render the weather statistics of the current location or a searched location. We learned how we should approach an application

development in React by starting with a mockup of the requirement, defining our UI component hierarchy using composition, identifying our state structure and tying up state with components to see our application come live. We can further enhance it and then build many more applications following the same approach. In the next chapter, we will be introduced to another way of defining and managing state in React applications using Redux.

Questions

1. Which external dependency is to be installed to make external API calls?

    A. lodash

    B. request

    C. axios

    D. Response

    Answer: Option C

    The Axios library is used to make HTTP calls.

2. The state of the component is being updated but the changes are not being rendered on the UI. What could be the reason?

    A. The state is mutated/changed directly by assignment.

    B. The element to the state array is added using push() function.

    C. The element of the state is removed using splice() function.

    D. All of the above

    Answer: Option D

    All the statements listed cause a mutating change which is not detected by React to re-render the component.

3. Which of the following is correct about package.json?

    A. This file is used to give the options about JS used for the React project.

    B. This file contains information about the React project and its dependencies.

    C. This file contains the system files required for React applications.

    D. All of the above.

    Answer: Option B

    Package.json contains information about the React project and its dependencies.

4. The _____ in class components can be used to handle the actions like API requests, manual DOM mutations, and logging.

   A. `constructor, getDerivedStateFromProps,`
   B. `componentDidMount, getDerivedStateFromProps`
   C. `useEffect hook, componentDidMount, componentDidUpdate`
   D. `useState hook, componentDidMount, componentDidUpdate`

   Answer: Option C

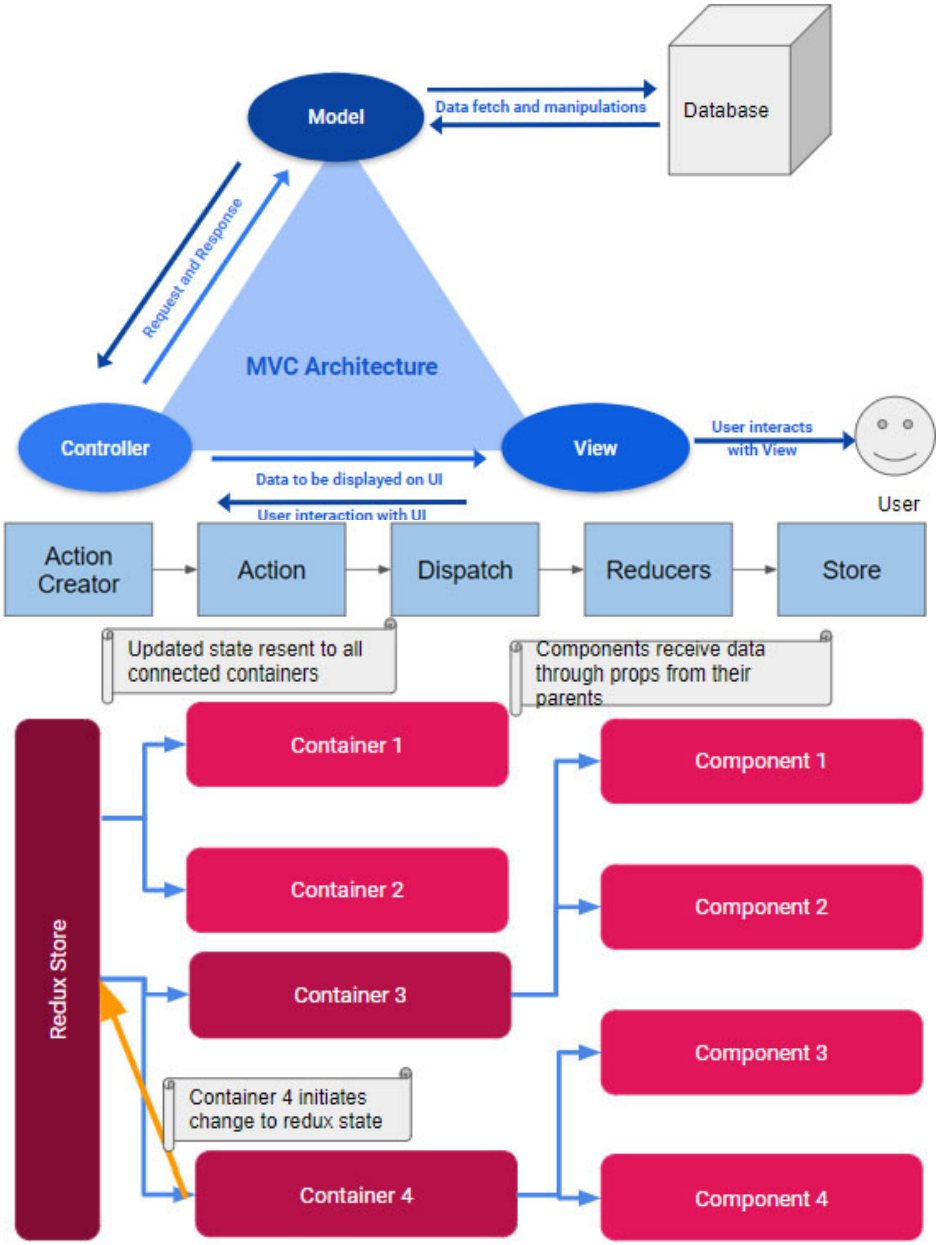
   The useEffect hook, componentDidMount, componentDidUpdatecan be used to handle the actions like API requests, manual DOM mutations, and logging.

5. Which of the following is true regarding the React state?

   A. The state should not be mutated/changed directly by assignment as React will not recognize the change in state and the changes will not be rendered.
   B. The state should be mutated/changed directly by assignment.
   C. For objects and arrays, the spread operator should not be used to make changes to state.
   D. Both A and C

   Answer: Option A

   The state should not be mutated/changed directly by assignment as React will not recognize the change in state and the changes will not be rendered.



CHAPTER 16

State Management in React Applications

In the last few chapters, we learned about React, how to build applications in React and how with the change in state and props, the React application is re-rendered. Data is an important aspect of any application. In this chapter, we will understand what state management is and how it can be done very predictably in React applications using Redux.

Structure

- State management
- The architecture of Redux
- Incorporating Redux in React

Objective

You will learn all about what an application state is and how to manage the state in React applications using Redux.

State management

First, let's understand what state is and what all this deal about state management is.

In any application, there is some data involved which changes as we interact with the application. Data could be the external data being fetched from APIs and later manipulated by the user or data being entered by the user or data indicating user login status, user behavior, and so on, but irrespective of what the application does, there will always be some data which the application deals with. Also, the application may behave differently based on the value of the state. For example:

- If a user is logged in, only then you can navigate to some secure pages.
- If the user language is French, show him content in French language, else show English language by default.
- Show the dashboard based on the user's default view preferences.
- There are multiple ways in which a state can impact application behavior.

Traditional applications followed the model-view-controller pattern to keep the data separated from the view and the logic layer.

In **Model-View-Controller (MVC), Model (M)** is the data of the application, **View (V)** is the user interface which is visible to the user, and **Controller (C)** comprises the logic to handle user interaction events and

update data accordingly.

In this approach, though there is a proper structure and clear segregation of logic and data layer, there is no certainty of the direction in which the data can update and the changes can flow. The direction of data flow and data changes flow is two ways, due to which it becomes difficult to manage and debug the data changes as shown in the following diagram:

Figure 16.1: *MVC architecture data flow*

The concept of state management defines the state of the application in the form of a data structure, which can be updated in a very predictable way in a single direction only. The unidirectional update to the state makes it very predictable to maintain and debug.

Facebook came up with a pattern for state management called Flux, along with some tools to implement Flux in React applications. Redux, inspired by Flux, is a full-fledged library for state management which is popularly used in React applications for state management. There are other ways and tools for state management like Flux, Mobx, Apollo Link State, which you can explore further if interested. In this book, we will look at state management using Redux.

So far, we understood the need for state management to handle the application state predictably. Now, we will see how Redux achieves this.

The architecture of Redux

"Redux is a predictable state container for JavaScript apps."

What does it mean when we say a predictable state container? What do we mean when we say that the data flow is unidirectional?

The state container maintains the data at one place in the form of a global store. Any changes, user events, and actions resulting in change of data always follow the same path in a single direction; hence, it is predictable, as shown in the following figure.

Let's consider the weather application, which we built in the previous chapter. Coordinates is one of the data elements which is part of the global state and can change depending on the user entry. So, we will see how to maintain the coordinates as a part of the store:

Figure 16.2: *Unidirectional data flow of the Redux architecture*

We need to understand each of these terms depicted in the preceding boxes which make up the Redux ecosystem.

To use Redux, we need to first install the dependency using the following command:

```
npm install redux --save
```

Action creator is the function which initiates or creates the action. The action creator to handle the update of coordinates will be as follows:

```
export constInitialCoordinates = (coordinates)=>{
 console.log(coordinates);
return{
type:'INITIAL_COORDS',
payload:coordinates,
 }
}
```

Action is the object returned by the action creator which represents the change to be done to the state. It has the following two parts:

- The `action` type, which indicates the type of change
- The `payload`, which carries the data to make the change

In the preceding example, the action includes the type as `INITIAL_COORDS` and the `payload` contains the changed coordinates as follows:

```
 {
type:'INITIAL_COORDS',
payload:coordinates,
      }
```

There can be multiple actions defined in an application to handle the different operations to be performed on the state.

Dispatch is the function which dispatches the action to the reducer along with the previous state. This is the base functionality of the dispatcher to ensure the state change action is passed to the reducer. This function can be wrapped by middleware to include additional capabilities like handling asynchronous functions, adding delay or wait, adding logging, or any other middleware functionality.

Dispatch is part of the Redux ecosystem and will be initiated using the dispatch() function from the Redux library.

Reducer is the most important part of the Redux cycle, which receives the action and previous state and returns the changed state based on the action type. It should not include any API calls. It only receives the action using which it determines the action type to be performed. Also, using the payload, it knows the changes to be applied. So, it applies the changes to the previous state and returns the changed state.

The reducer function looks something like the following code that handles only the `INITIAL_COORDS` action now:

```
export default (state = initialState, action) => {
 switch (action.type) {
```

```
case'INITIAL_COORDS':
      // console.log(action.payload);
return{
       …state,coordinates:action.payload
      }
      default:
return{
       …state
      }
 }
```

The store is an object which handles the complete application state at the global level. There will be a single instance of a store which will contain all the different pieces of state for the application. This is what is updated by the reducer and this is where the latest state will always be.

The store provides the following capabilities:

- Holds the application state
- Allows access to the state using the `getState()` function
- Allows the state to be updated using the `dispatch(action)` function to dispatch the action for a specific update

The store can be created using the `createStore` function from the Redux library as follows:

```
createStore(reducer, [preloadedState], [enhancer])
```

The arguments are as follows:

- `reducer (Function)`: A function that takes the current state tree and applies an action to return the reduced/changed state tree.
- `[preloadedState] (any)`: The initial state to initialize the store.
- `[enhancer] (Function)`: The store enhancer. You may optionally specify it to enhance the store with third-party capabilities such as middleware, time travel, persistence, and so on. The only store enhancer that is shipped with Redux is `applyMiddleware()`.

Middleware is some code logic which can be placed in between the point a framework receives a request and the point at which it generates a response. Redux also provides the middleware capability to handle the point between dispatching an action and the point at which it reaches the reducer.

Redux middleware is useful for logging, getting data from an asynchronous API, routing, and so on. The method `applyMiddleware()` is the only store enhancer which can be used to combine different middleware. It is available in the Redux basic package and can be used to apply more than

one middleware at the time of creating the store.

Why is middleware needed?

If we fetch data and the response needs to be dispatched to the reducer, the fetch may be an asynchronous call, then by the time dispatch is called, you may not have received the response from your API; hence, the need of middleware.

We will be using the third-party library `redux-thunk` in our example to include middleware, mainly to handle asynchronous API calls.

When using `redux-thunk`, the action creators can return two types of responses:

- The standard action object with type and payload
- A function which is invoked with dispatch and the action is dispatched manually in the function after the asynchronous response is received

Putting all these pieces together, the Redux flow starts with the action creator initiating an action to make some changes to the state with the action type and payload. The action is dispatched to the reducer along with the previous state. The reducer does the change based on the type and payload and returns the changed state. So, any change in state will always follow the same path and it becomes very predictable to follow.

With this basic understanding of how Redux works, next we will see how Redux works with React.

Incorporating Redux in React

In React, we saw that the data could be handled in two ways: the components can have a local state of their own, and the data can be passed on to child components in the form of props.

If the data has to be shared across the different components of the application, and all changes to the data need to be handled in a unidirectional and predictable manner, then Redux can be used for state management. Apart from state management, Redux is a rock-solid framework which also provides other add-on capabilities like middleware.

The presentation logic of the React components can be differentiated based on whether or not they connect to the Redux store. They are categorized as components and containers as follows:

	Components	Containers
Concerns	How components look (markup, styles)	How components work (data fetching, state updates)
Aware of Redux	No	Yes
To read data	Read data from props	Subscribe to Redux state
To change data	Invoke callbacks from props	Dispatch Redux actions
Are written	By hand	Usually generated by React Redux

These actions defined to dispatch actions using the props	These actions defined to dispatch actions as the reducer to make changes to the state.
View will be rendered only the redux store contains some	View will be rendered only when the redux state changes
	changes, the state view will receive the latest updated data.

Table 16.1: *Components versus containers*

The following diagram shows some containers connected to the Redux store and some other components which are not connected to the store directly:

Figure 16.3: *Components and containers in React*

In order to use Redux with React in our application, we will need some additional dependencies:

- `Redux`: The Redux framework.
- `React-redux`: The library which makes it easier to use Redux from React and provides different functions for the Redux functionality to be incorporated in React application.

The Redux terminologies defined earlier hold well when working in the React environment, but we need some additional capabilities to be able to replicate the following steps in a React application:

- Create and initialize the Redux store along with the capability to handle any asynchronous middleware operations.
- Define action creators and actions as part of the Redux definition.
- Dispatcher to dispatch actions to the reducer.
- Define reducers which will handle the actions and update the store.
- React components to be able to connect to the store to get the state or be able to dispatch actions.
- Capability to handle middleware for asynchronous API calls.

To get started with Redux in a React application, you first need to add the new dependencies as follows:

```
npm install react-redux redux redux-thunk --save
```

Set up and provide the Redux store to the React application

After having installed the dependencies, you will first define the store structure and set up the store with the initial state and the reducers using the `createStore` function from the `redux` library as follows:

```
import App from './App';
import thunk from 'redux-thunk';
import reducer from './reducer';
```

```
import { createStore,applyMiddleware,compose } from
'redux';

const
store=createStore(reducer,composeEnhancers(applyMiddleware
```

`redux-thunk` is included for middleware in order to be able to make asynchronous calls in the application.

Once the store has been set up, it needs to be provided to the React application to be available for the components in the application. This is done using the `Provider` component of the `react-redux` library as shown in the following code:

```
import { Provider } from 'react-redux';
ReactDOM.render(
<Provider store={store}>
<App/>
</Provider>, document.getElementById('root'));
```

The preceding code snippets are included in the `index.js` file where we have the logic to render to the DOM. This is where the `Provider` is a wrapper around the root component `App` and `store` provided for the entire application.

The other aspects of Redux will hold the same and the action creators, actions, and reducers will have to be defined.

React components connect to the Redux store

Once the Redux store is set up with the initial state, action creators, actions and reducers, and the store is provided to the React applications, the React components need capabilities to be able to connect to the store.

What will they need from the Redux store?

1. Parts of data from the state maintained in the Redux store.
2. Capability to dispatch actions to the reducer to be able to make changes to the state in Redux.

The `react-redux` library provides the connect function which connects the React component to the store.

Let's understand how the connect function helps achieve the two required capabilities to the React component.

The connect function accepts the following parameters; all of which are optional and have the following names by convention. You can use any names but using the following names make your code consistent and easy to understand:

- **mapStateToProps?function:**
  `mapStateToProps?: (state, ownProps?) => Object`

  This first parameter is an optional parameter of a type function, which you can provide if you want to make use of the state from the Redux store in your component.

  Let's look at how the function `mapStateToProps` can be used to work with Redux store as listed below:

  - This function receives the global state from Redux as an input and returns an object which gets merged into the props of the component. Anytime the global state changes, this function is called and if the cause of the change further results in a change in its output, that is, changes in the props of the component, the React component will be re-rendered.
  - This function can also receive an additional second parameter, conventionally called `ownProps`, which will contain the props of the wrapped component. This may be useful if the state computations are dependent on the prop values of the component in consideration.
  - Finally, the function returns an object which becomes available for the component as its props. Whenever this object changes, the component will be re-rendered:

```
constmapStateToProps=(state)=>{
return{
state:state.coordinates,
 }
}
export default connect(mapStateToProps)(App);
```

Whenever the coordinates change in the global state, the App component will be re-rendered.

- **mapDispatchToProps? function | object**

```
mapDispatchToProps?: Object
mapDispatchToProps?:(dispatch, ownProps?) => Object
```

  This optional second parameter can be a function which returns an object or an object directly. It is provided when you want to dispatch any of the actions to the reducers from your component.

  The parameters for this method have the following behavior:

  - If this parameter is not provided, your component will receive the dispatch function in props using which you can dispatch the actions to the reducer.
  - If this parameter is defined as a function, it receives dispatch as

a first argument and returns an object which includes all the actions the current component needs to dispatch.

```
constmapDispatchToProps = dispatch => {
 return {
  // dispatching actions returned by action
  creators
  first: () => dispatch(first()),
  second: () => dispatch(second())
 }
}
```

The returned object gets included in props so these action creators can be directly invoked without explicitly dispatching within the component.

- This function can make use of the bindActionCreators function provided by Redux which will not require explicit binding with the dispatch function as follows:

```
import { bindActionCreators } from 'redux'

function mapDispatchToProps(dispatch) {
 return bindActionCreators({ first, second
 }, dispatch)
}
```

The first argument is the list of action creators in the form of an object which needs to be dispatched within the function:

- This can be an object which includes the list of action creators to be dispatched. In this case, it will automatically take care of binding with dispatch.

```
export default connect(mapStateToProps,
{SearchAction,InitialCoordinates})(App);
```

App component can dispatch the actions for `SearchAction` and `InitialCoordinates`.

- **mergeProps? function:**

```
mergeProps?: (stateProps, dispatchProps, ownProps)
=> Object
```

This optional parameter explicitly specifies how the final merged object should be. If not specified, it will be as follows:

  - { …ownProps, …stateProps, …dispatchProps } by default
  - You can specify a different set and order using this parameter.

- **options? object:**

The last parameter which is an optional object gives an additional

capability to control how the comparison of data is done when the global state is received. It has the following options:

```
{
context?: Object,
pure?:boolean,
areStatesEqual?: Function,
areOwnPropsEqual?: Function,
areStatePropsEqual?: Function,
areMergedPropsEqual?: Function,
forwardRef?:boolean,
}
```

Using the `connect` function and its optional parameters, you can control how your component wants to connect to the store and what it wants to extract and use from the Redux store.

Having been introduced to all aspects of redux and how it can be used in a React application, you will now convert the weather application, developed in *Chapter 15, Building an application with React*, into using Redux to maintain the global state.

This example is included for understanding the different aspects of coding a React application with Redux.

Let's first define all the actions to be handled. As part of the action creator, you will also see some functions to handle the API calls.

The `actions/index.js` file will contain the following action creator definitions:

```
//To initialise the coordinates to current location of
user
export constInitialCoordinates = (coordinates)=>{
 console.log(coordinates);
return{
type:'INITIAL_COORDS',
payload:coordinates,
 }
}
//To get the latest coordinates based on user search
term - Async call handled by function which dispatches
action manually
export constSearchAction = (city) => {
 let locality = city,
countryRegion = 'IN',
   coordinates={};
  return dispatch => {
```

```
    return axios.get(`http://dev.virtualearth.net/REST/
    v1/Locations?&countryRegion=${countryRegion}
    &locality=${locality}&key=
    ${geocoding_Api_key}`).then((response) => {
     let locationCoordinates =
     response.data.resourceSets.map((data) => {
      return (
data.resources.map((resourceData) => {
       if (resourceData.confidence === "High") {
        coordinates={
         latitude: resourceData.point.coordinates[0],
longitude :resourceData.point.coordinates[1]
       };
       return null;
       }
       else
       return null;
      })
     )
     })
    dispatch(SearchActionCreator(coordinates))
   })
 }
}
// To update coordinates once received from API
response
constSearchActionCreator = (responseData) => {
 return {
 type: 'SEARCH_ACTION',
 payload: responseData
 }
};

//To update the search term as entered by user in
global state
export constSearchTermUpdate = (term) => {
 return {
   type: 'TERM_UPDATE',
   payload: term
 }
 }
 //To fetch the main weather data from API
export constMainWeatherData = (data) => {
 return dispatch =>{
    return axios.get(`https://api.openweathermap.org/
```

```
    data/2.5/weather?lat=${data.latitude}&lon=
    ${data.longitude}&units=metric&appid=
    ${Api_key}’).then((response) => {
        dispatch(MainWeatherDataAction(response.data))
       })
.catch(err => {
         console.log(err);
        });
 }
}

//To Update the main weather data
constMainWeatherDataAction = (responseData)=>{
return{
type:"WEATHER_DATA",
payload:responseData
 }
}

//To fetch the data for the WeatherChart component
export constWeatherChartAction = (data) => {
 return dispatch =>{
     return axios.get(‘https://api.openweathermap.org/
     data/2.5/forecast?lat=${data.latitude}&lon=
     ${data.longitude}&units=metric&cnt=7&appid=
     ${Api_key}’).then((response) => {
      // console.log(response.data);
      dispatch(ChartAction(response.data))
     })
.catch(err => {
       console.log(err);
       });
 }
}
//To update the Weather Chart data
constChartAction = (responseData)=>{
return{
type:"CHART_DATA",
payload:responseData
 }
}

//To fetch the forecast UV index data from API
export constTabularDataAction = (data) => {
 return dispatch =>{
     return axios.get(‘http://api.openweathermap.org/
```

```
      data/2.5/uvi/forecast?appid=${Api_key}&lat=
      ${data.latitude}&lon=${data.longitude}
      &cnt=8').then((response) => {
       // console.log(response.data);
       dispatch(TabularAction(response.data))
      })
.catch(err => {
       console.log(err);
       });
 }
}

//To update the Forecast UV Index tabular data
constTabularAction = (responseData)=>{
return{
type:"TABULAR_DATA",
payload:responseData
 }
}
```

The initial global state will be defined as follows as part of reducers/`index.js`:

```
constinitialState={
coordinates:{
 },
searchTerm: ""
}
```

The reducers will be defined as follows to handle the different actions in reducers/`index.js`:

```
export default (state = initialState, action) => {
 switch (action.type) {
case'INITIAL_COORDS':
      // To set the coordinates to current location of
      user
return{
       …state,coordinates:action.payload
      }
      case 'SEARCH_ACTION':
      // To set the coordinates based on the searched
      location
      return {
       …state,
coordinates:action.payload
      }
```

```
      case 'TERM_UPDATE':
      //To update the search term as entered by the user
      return {
       …state,
searchTerm:action.payload
      }
      case 'WEATHER_DATA':
      // To maintain weather data in store
      return {
       …state,
mainWeatherData:action.payload
      }
      case 'CHART_DATA':
      // To maintain chart related data in store
return{
       …state,
chartData:action.payload
      }
      case 'TABULAR_DATA':
      // To maintain Table related data in store
return{
       …state,
tabularData:action.payload
      }
      default:
return{
       …state
      }
 }
}
```

Next, update the `index.js` to create and provide the store to the application:

```
constcomposeEnhancers =
window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ || compose;
const
store=createStore(reducer,composeEnhancers(applyMiddleware

ReactDOM.render(
<Provider store={store}>
<App/>
</Provider>, document.getElementById('root'));
```

Now, the store is created and is available for the `App` component. The `App` component will not have any local state. It will get data from the store

and update the store by dispatching actions:

```
let newState = {}
class App extends Component {

 //Method to get current location of user
componentDidMount(){
navigator.geolocation.getCurrentPosition(position => {
constcoords = position.coords;
newState = {
      latitude: coords.latitude,
      longitude: coords.longitude,
    }
this.props.InitialCoordinates(newState);// dispatch
action to update current location coordinates
 });
 }

 //Method which handle the change of input and set it
 to state
handleChange=(e) => {
this.props.SearchTermUpdate(e.target.value);
 }

 //Method which set the state of searchTerm with input
 value onclick of search button
handleSearch=() => {
 if (this.props.searchTerm === "Current location") {
this.props.InitialCoordinates(newState)
 }
 else {
this.props.SearchAction(this.props.searchTerm);
 }
 }

 //render method
render() {
 if (this.props.coordinates !== undefined) {
   return (
<div className="App">
<Header handleSearch={() =>this.handleSearch()} //
Header Component
handleChange={(e) =>{ this.handleChange(e) }}
        />
<div className="container">
<MainData/> {/* Main Component which consists
```

```
cards,Summary weather table,Weather Chart*/ }
</div>
<TabularData/> {/*which consists Uv index tabular data
for 8days ahead */}
</div>
      );
 }
 else
   return(<div>
<Header handleSearch={() =>this.handleSearch()}
handleChange={(e) =>{ this.handleChange(e) }}
        />
<p>Loader…</p>
</div>)
 }
}
constmapStateToProps=(state)=>{
return{
coordinates:state.coordinates,
searchTerm: state.searchTerm
 }
}

export default connect(mapStateToProps,
{SearchAction,InitialCoordinates, SearchTermUpdate})
(App);
```

Now, the App component does not send the coordinates as `props` to the `MainData` and `TabularData` components as it can connect to the store and get the coordinates.

Let's look at `MainData` which will connect to the store to get the data and send the required data to its child components: `WeatherCard` and `SummaryTable`. The third child `WeatherChart` requires extra logic so it will directly connect to the store to get the coordinates:

```
constMainData=(props)=>{
 // console.log(props.coords)
 // const {latitude, longitude} = props.coordinates;

useEffect(() => {
   if(props.coordinates)
props.MainWeatherData(props.coordinates);
},[props.coordinates])

if(props.weatherData&&
props.weatherData!==undefined){
```

```
return(
<div className="main_container">
<div className="card_container_main">
<WeatherCardweatherData={props.weatherData}
mainData={props.mainData}
windData={props.windData} />
<Weatherchart/>
</div>
<SummaryTableweatherData={props.weatherData}
mainData={props.mainData}
windData={props.windData}
sunData={props.sunData}
cloudData={props.cloudData}
          coordinates={props.coordinates}/>
</div>
 )
}else{
return(
<div></div>
 )
 }
}
constmapStateToProps=(state)=>{
 if(state.mainWeatherData!==undefined){
 console.log(state.mainWeatherData.weather[0])
return{
coordinates :state.coordinates,
weatherData: state.mainWeatherData,
mainData:state.mainWeatherData.main,
windData:state.mainWeatherData.wind,
sunData:state.mainWeatherData.sys,
cloudData:state.mainWeatherData.weather[0]
 }
}
else{
return{
coordinates :state.coordinates
 }
}
}
export default connect(mapStateToProps,
{MainWeatherData})(MainData);
```

The child components `WeatherCard` and `SummaryTable` are unchanged as they do not connect to the store but get the data from their

parent `MainData`.

The third child `WeatherChart` will now directly connect to the store and get the coordinates and also dispatch the action to generate the chart-related data.

The `WeatherChart` component is changed to now connect to the store directly and use coordinates; the changes (in bold) look like the following code:

```
constWeatherchart = (props)=>{
useEffect(() => {
props.WeatherChartAction(props.coordinates)
},[props.coordinates]);
if(props.chartData !==undefined &&
props.chartData.list!==null){
varapiData={
labels:props.chartData.list.map((data)=>{
       let time = data.dt_txt.slice(11,16);
       let Hours = +time.substr(0, 2);
       let hour = (Hours % 12) || 12;
       hour = (hour < 10)?("0"+hour):hour;
       let ampm = Hours <12 ? " AM" : " PM";
       time = hour + time.substr(2, 3) + ampm;
       return time;
        }),
       datasets: [
        {
type:'line',
       label: 'Temperature for every 3hours',
data:props.chartData.list.map((data)=>{
         return data.main.temp }),
backgroundColor :'rgba(0,0,0,0)',
borderColor :'#475eb2',
pointBackgroundColor:'#475eb2'
        },
        {
type:'bar',
label:'Humidity for every 3hours',
       fill: false,
lineTension: 0.1,
backgroundColor: 'rgba(75,192,180,0.4)',
borderColor: 'rgba(75,192,192,1)',
barPercentage: 0.2,
data:props.chartData.list.map((data)=>{
        return data.main.humidity }),
```

```
        }
         ]
       }
    }
const options = {
      legend: {
       display: true,
      },
    };
 return (
<div className="weather_container">
<div style={styles.graphContainer}>
<Bar
className="weather_chart"
      data={apiData}
      options={options}
       />
</div>
</div>
 )
 }
constmapStateToProps = (state) =>{
return{
chartData:state.chartData,
 coordinates: state.coordinates
 }
}
export default connect(mapStateToProps,
{WeatherChartAction})(Weatherchart);
```

Also, the `TabularData` component is changed in the same way to connect to the store directly and get the coordinates instead of getting the props from the parent.

The `TabularData` component code looks like the following code (changes in bold):

```
constTabularData = (props) => {
useEffect(()=>{
props.TabularDataAction(props.coordinates)
 }, [props.coordinates]);
if(props.tabularData !==undefined
&&props.tabularData!=={}){
return(
<div>
<h3 className="header">Forecast UV Index data</h3>
```

```
<div className="table_data">
<table >
<thead>
<tr>
<th>Day</th>
        {props.tabularData.map((data,i)=>{
return(
<th key={i}>{data.date_iso.slice(0,10)}</th>
        )})}
</tr>
</thead>
<tbody>
<tr>
<td>Uv value</td>
        {props.tabularData.map((data,index)=>{
return(
<td key={index}>{data.value}</td>
       )})}
</tr>
</tbody>
</table>
</div>
</div>
 )
 }
else{
return(
<div></div>
    )
 }
}
constmapStateToProps=(state)=>{
return{
  coordinates: state.coordinates,
tabularData: state.tabularData
  }
}

export default connect(mapStateToProps,
{TabularDataAction})(TabularData);
```

The functionality and UI remains the same but now the state is maintained at a global store level. Any changes are propagated from the store to all the components connected to the store.

You can check the complete code bundle accompanied with this chapter

showing the React application making use of the Redux store.

Conclusion

In this chapter, we learned about Redux and how it can be used to manage the global state along with middleware to handle asynchronous API calls in our React applications. In the next chapter, we will cover the other aspects of React application development, namely, debugging, testing, and deploying.

Questions

1. The whole state of your app is stored in an object tree inside a single____. The only way to change the state tree is to emit an ____. To specify how it can transform the state tree, you write pure ____.

   A. state, action creator, functions

   B. store, action, reducers

   C. store, change, functions

   D. state, dispatch, reducers

   Answer: Option B.

2. ____ is the enhancer for the store used to combine different middleware, available in the ___ package and can be used to apply more than one middleware.

   A. applyMiddleware, redux

   B. applyMiddleware, react-redux

   C. createStore, redux

   D. createStore, react-redux

   Answer: Option A.

3. The store has the following responsibilities:

   i. Holds the application state

   ii. Allows access to the state via getState()

   iii. Allows the state to be updated via dispatch(action)

   iv. Define action creators.

   Which are the valid options from the above list?

   A. i, ii

   B. i, ii, iv

   C. i, ii, iii

D. All of the above

Answer: Option C.

4. If you do not supply your own **`mapDispatchToProps`** function or object full of action creators, the default `mapDispatchToProps` implementation will

   A. Not add any props into your component's props

   B. Inject dispatch into your component's props.

   C. Be ignored

   D. None of the above

   Answer: Option B.

5. To create a redux store as below->createStore(reducer, [preloadedState], [_____]), what is the third parameter?

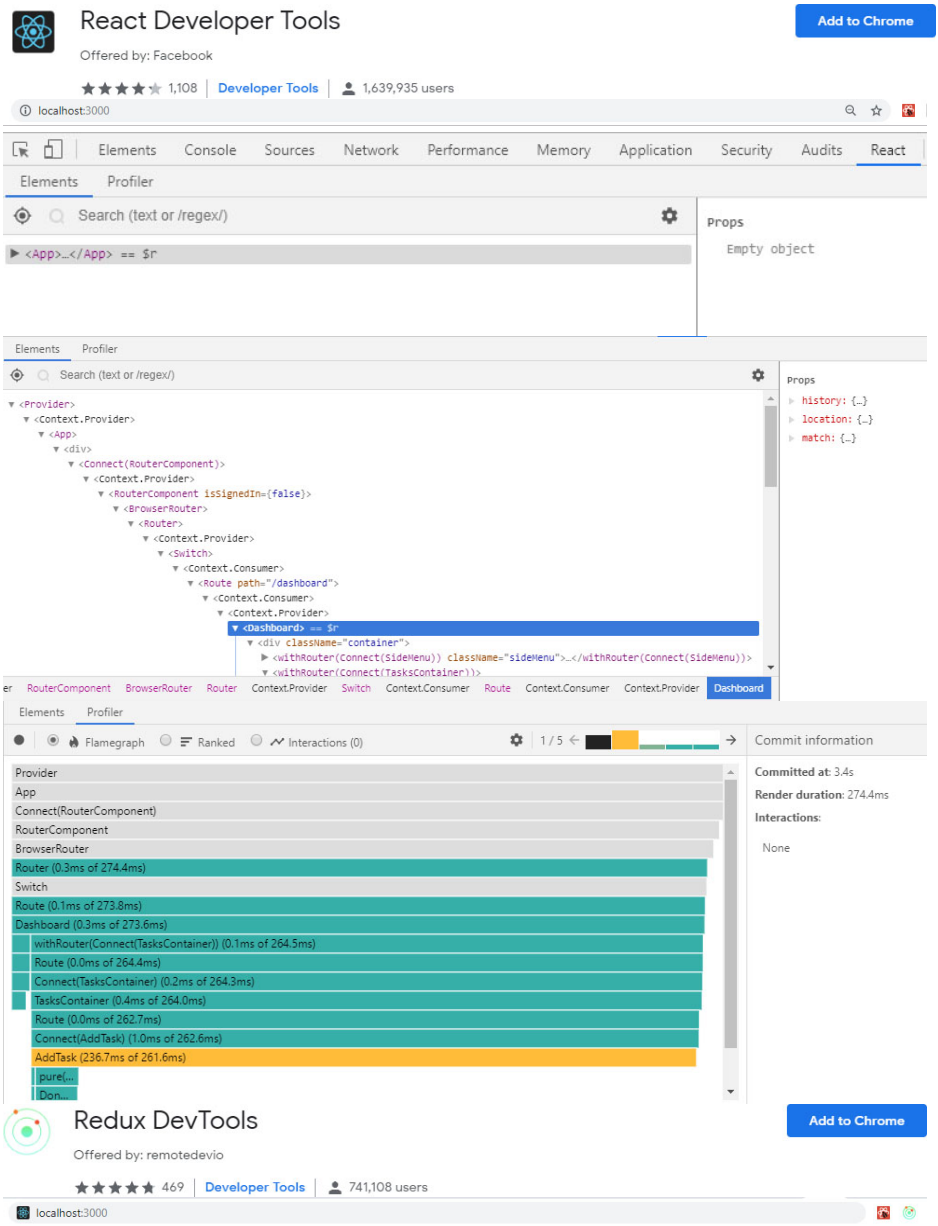
   A. persistence

   B. fragment

   C. component

   D. enhancer

   Answer: Option D


React Developer Tools
Offered by: Facebook
Add to Chrome
1,108 | Developer Tools | 1,639,935 users
localhost:3000
Elements Console Sources Network Performance Memory Application Security Audits React
Elements Profiler
Search (text or /regex/)
<App>…</App> == $r
Props
Empty object
Elements Profiler
Search (text or /regex/)
<Provider>
<Context.Provider>
<App>

<Connect(RouterComponent)>
<Context.Provider>
<RouterComponent isSignedIn={false}>
<BrowserRouter>
<Router>
<Context.Provider>
<Switch>
<Context.Consumer>
<Route path="/dashboard">
<Context.Consumer>
<Context.Provider>
<Dashboard> == $r

<withRouter(Connect(SideMenu)) className="sideMenu">…</withRouter(Connect(SideMenu))>
<withRouter(Connect(TasksContainer))>
Props
history: {…}
location: {…}
match: {…}
RouterComponent BrowserRouter Router Context.Provider Switch Context.Consumer Route Context.Consumer Context.Provider Dashboard
Elements Profiler
Flamegraph Ranked Interactions (0)
1 / 5
Commit information
Provider
App
Connect(RouterComponent)
RouterComponent
BrowserRouter
Router (0.3ms of 274.4ms)
Switch
Route (0.1ms of 273.8ms)
Dashboard (0.3ms of 273.6ms)
withRouter(Connect(TasksContainer)) (0.1ms of 264.5ms)
Route (0.0ms of 264.4ms)
Connect(TasksContainer) (0.2ms of 264.3ms)
TasksContainer (0.4ms of 264.0ms)
Route (0.0ms of 262.7ms)
Connect(AddTask) (1.0ms of 262.6ms)
AddTask (236.7ms of 261.6ms)
Committed at: 3.4s
Render duration: 274.4ms
Interactions:
None
Redux DevTools
Offered by: remotedevio
Add to Chrome
469 | Developer Tools | 741,108 users
localhost:3000

```
FAIL src/components/__tests__/addTask-test.js
  The testcase set
    √ sample test case to be tested to pass  (6ms)
    × sample test case to fail (13ms)

  ● The testcase set › sample test case to fail

    expect(received).toBeTruthy()

    Received: false

       8 |
       9 |   it('sample test case to fail', () => {
    > 10 |     expect(false).toBeTruthy();//false cannot be truthy so it should fail
         |                   ^
      11 |   })
      12 | })
      13 |

      at Object.toBeTruthy (src/components/__tests__/addTask-test.js:10:19)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 passed, 2 total
Snapshots:   0 total
Time:        5.845s, estimated 6s
Ran all test suites.
FAIL src/App.test.js
  ● App › should render App component

    expect(value).toMatchSnapshot()

    Received value does not match stored snapshot "App should render App component 1".

> todo_app@0.1.0 build C:\xampp\htdocs\ToDoApp-master
> react-scripts build

Creating an optimized production build...
Compiled with warnings.

./src/components/myProfile.js
  Line 36:  Unreachable code  no-unreachable

./src/components/Taskitem.js
  Line 46:    Do not mutate state directly. Use setState()            react/no-direct-mutation-state
  Line 98:   Expected to return a value in arrow function            array-callback-return
  Line 128:  Expected to return a value at the end of arrow function  array-callback-return
  Line 271:  Expected to return a value at the end of arrow function  array-callback-return

Search for the keywords to learn more about each warning.
To ignore, add // eslint-disable-next-line to the line before.

File sizes after gzip:

  133.81 KB        build\static\js\2.b19c9fbf.chunk.js
  16.86 KB (-1 B)  build\static\js\main.c60aff72.chunk.js
  2.62 KB          build\static\css\main.90077376.chunk.css
  762 B            build\static\js\runtime~main.a8a9905a.js
  676 B            build\static\css\2.16cfc387.chunk.css

The project was built assuming it is hosted at the server root.
You can control this with the homepage field in your package.json.
For example, add this to build it for GitHub Pages:

  "homepage" : "http://myname.github.io/myapp",

The build folder is ready to be deployed.
You may serve it with a static server:

  npm install -g serve
  serve -s build

Find out more about deployment here:

  https://bit.ly/CRA-deploy
```

Debugging, Testing, and Deploying React

“Quality is never an accident; it is always the result of intelligent effort.”

– John Ruskin

In the last few chapters, we learned how to build applications in React and how to manage the state of React applications using Redux. During any kind of software development, errors are evident and we should be able to debug them in the right manner to be able to solve the errors efficiently. Automation testing is another important aspect to ensure good quality deliverables. Earlier, we had seen how debugging, testing, and deployment can be done for JavaScript applications.

In this chapter, we will focus on these important aspects with respect to a React application development which includes debugging, testing, and deploying the application. We will learn about the tools which can be used for debugging and automation testing. We will also learn about the process of deploying React applications.

Structure

- Debugging React applications
- Testing React applications
- Deploying React applications

Objective

At the end of this chapter, you will learn how to debug React applications effectively. You will also be introduced to unit testing automation for React apps and how to build them for production deployment.

Debugging React applications

In *Chapter 10, Debugging JavaScript Applications*, we explored the Chrome devtools and its capabilities which are used in debugging JavaScript applications. Chrome devtools, or any corresponding Browser devtools is the basis of debugging in any browser-based application. All the features of Chrome devtools are applicable and useful for a React application as well.

When debugging a React application in your browser, you will inspect and

look at the console for errors and the other tabs to investigate the issue and debug your application like any other JavaScript application.

In addition to this, there are some extensions available which further enhance and improve the debugging experience making it much more effective with respect to React.

Reactdeveloper tools

React developer tools is a browser plugin available to debug and investigate React applications in detail. It is available as a plugin for Chrome and Firefox browsers and also available as a standalone application for debugging in Safari browser and React Native applications.

Let's see how we can use this plugin to debug React applications:

STEP 1:

Install the plugin or the standalone application depending on the browser of your choice. If you have been using the Chrome browser along with this book, go ahead and install the Chrome devtools extension from the Chrome web tools as shown in the following screenshot:

Figure 17.1: *React Developer Tools in Chrome Web Store*

STEP 2:

Once installed using the **`Add to Chrome`** button, it will appear as an indicator in your Chrome browser using which you can know if the extension is active for the current site or not as shown (in the active state):

Figure 17.2: *Browser React developer tools extension icon*

STEP 3:

It gets added as an additional tab called **`React`** in the Chrome Developer tools as shown in the following screenshot:

Figure 17.3: *Chrome devtools with React devtools extension as an additional tab*

STEP 4:

On exploring the React devtools further, you see that it has two tabs:

- Elements
- Profiler

The **`Elements`** tab shows the DOM tree like the main elements tab, but the difference is that this HTML tree is made up of JSX elements. It will show the JSX elements which make up the React application. On selecting a specific element, you get a complete view of the props and the state

values associated with the element in the section on the right side. You can also change the values and see the changes reflected in the application on the fly. The **`Elements`** tab looks like the following screenshot with the complete JSX structure of your application:

Figure 17.4: *Elements tab of React devtools*

The **`$r`** represents the current active node being inspected which can be rendered in the console.log using this representation.

The **`Profiler`** tab works similar to the JavaScript profiler where you can record the performance of a set of interactions with the application. You can run the profiler and perform the interactions with the application to record the performance.

It shows the profile as shown below for the different renders that happen and the time taken for each with a color coded flame graph as shown in *Figure17.5:*

Figure 17.5: *React devtools Profiler sample output*

Using React devtools in conjunction with the Chrome browser tools gives a complete structural view of the application and makes it easier to delve into and spot issues and bugs.

Redux devtools

If you are using Redux for state management in your React application, there is another useful browser devtools extension which comes very handy to debug applications using Redux. Just like React devtools, Redux devtools is also available as a plugin for Chrome and Firefox browsers and as a standalone application for debugging in Safari browser and React native applications. These plugins make debugging easy by giving a peak into the program as the execution happens showing the values held in local state, redux store, etc.

In order to use this plugin in your React-Redux application, you can follow the given steps:

STEP 1:

This extension can be added from the chrome web store to the Chrome browser as shown in the following screenshot:

Figure 17.6: *Redux devtools at Chrome web store*

STEP 2:

Now, for a React-Redux application, the browser shows both icons active, as shown in the following screenshot:

Figure 17.7: *React and Redux devtools active for your application*

STEP 3:

For the Redux devtools to work there are some additional steps that need to be performed after the extension is installed, which include the following:

- Include the following code in `index.js` where the store is being created:

```
const composeEnhancers =
window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ ||
compose;
```

- Create the redux store using the `composeEnhancers`, created in previous step if no middleware is needed:

```
const store = createStore(
 Reducer,
 composeEnhancers())
);
```

Or as below if middleware is needed:

```
const store = createStore(
 Reducer,
 composeEnhancers(applyMiddleware(thunk))
);
```

STEP 4:

Now, you can navigate to the Redux devtools as the new tab in the Chrome browser devtools just like the React devtools.

1. Redux devtools shows the different actions which get dispatched from the beginning of the application initiation, the store value after every action, the change in store value, and so on.

   It gives a complete view of how the store changes between the different actions as shown in the following screenshot:

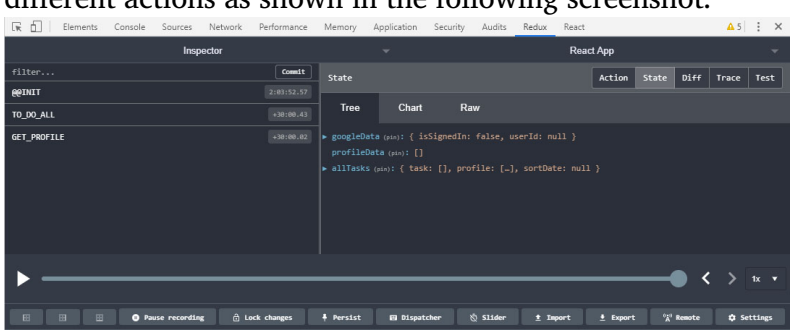


Figure 17.8: *Redux devtools Inspector view*

2. The left panel shows the different actions which were dispatched. On selecting a specific action, the right panel shows the following details:
    - **`The Action`** tab shows the details of the action, including type and payload.
    - The **`State`** tab shows the store details.
    - The **`Diff`** tab shows the variations or changes in the store between different actions.
    - **`Trace`** and **`Test`** tabs are used to trace and test the application after enabling these functionalities.

The Redux tools give a good view which is helpful in debugging to check:

- If the action got dispatched and with a proper type and payload
- If the action got dispatched in the correct order
- If the store got initialized correctly
- If the store got updated correctly as expected and so on

Using these two extensions, you can look into the application during the various interactions and investigate the app state (store) and props in detail to locate the problem and debug it easily and quickly.

Testing React applications

After having debugged your application and fixed your problem, now your application is ready to be tested. As you were introduced to testing and automation in *Chapter 11, Unit Testing Automation*, testing is an extremely important step in the development process and you should strive to automate your testing to increase its efficiency and effectiveness.

To reiterate the importance of a strong testing setup in your React application, remember the benefits it brings along:

- Properly written test cases provide a good understanding of what the component does and serve as a supporting document.
- Helps in discovering bugs and issues early in the development phase.
- Promotes writing good well-structured and modular code which can be tested easily.
- Reduces the risk of missed bugs when the application undergoes change and refactoring.

In this section, you will be introduced to some of the tools used to automate unit testing for React components:

- **Jest:** This is a JavaScript testing framework which is commonly used for testing React components. Jest is already configured if you use

create-react-app to create your application.

The benefits of using Jest over any other test runner include the following:

- Jest has very good speed and performance.
- Jest uses Snapshot testing to save the UI as a snapshot for the first time a component is tested and then report any changes to the snapshot. If changes are accepted, it becomes the latest snapshot to be used for comparison next time.
- **Enzyme:** This is a React-specific testing library which provides special capabilities to test React components. Testing can be automated without Enzyme, but with Enzyme, it becomes simpler and easier to use by providing utilities to enable shallow rendering, DOM testing, and rendering.

Test file organization

Test files are the test scripts which are created to test the application. The component-related test files are placed next to the component itself to maintain the ease of proximity. Jest will use the following convention to locate the test files:

- With `.js suffix in __tests__ folders.`
- With `.test.js suffix.`
- With `.spec.js suffix.`

We can use the unit testing file as `componentName.test.js` to be present in the same folder as `componentName.js` for the test logic.

Let's now see how we can test using Jest.

Defining the tests

The syntax to assert the test cases is similar to what we saw in *Chapter 11, Unit Testing Automation*.

The following example shows two dummy test cases which are expected to pass and fail based on the values being passed as `true` and `false`:

```
describe('The test case set', () => {
 it('sample test case to be tested to pass ', () => {
  expect(true).toBeTruthy();
 })
 it('sample test case to fail', () => {
  expect(false).toBeTruthy();//false cannot be truthy
  so it should fail
 })
})
```

This sample test script can be saved following the convention as defined earlier (in this example, I saved it as `add-task-test.js` under `__tests__` folder), and you can run it using the following command at the command prompt:

```
npm test
```

The result of the above test case is shown in the following screenshot which shows as 1 test being passed and second one failed:

Figure 17.9: *Test Result for sample test suite*

Some main expectations which can be used to compare and check the results include:

- `toBeTruthy()`
- `toBeFalsy()`
- `toBe()`
- `toEqual()`
- `toBeDefined()`
- `toBeCalled()`

You can frame your test cases using the preceding list of expectations to compare the expected with the actual results.

Snapshot testing

The snapshot testing approach saves a snapshot of the component being rendered the first time. Thereafter, every time it compares any changes in the snapshot to make sure the change is intentional and verified by the developer and not accidental. This way the rendered component can be validated for any changes done, and if the changes are accepted by the developer, the snapshot is updated with the latest changes.

We will use the react-test-renderer to render the component like objects without depending on the DOM or other environments to enable comparing snapshots of the component. In order to perform snapshot testing, we can follow the below steps:

STEP 1:

The dependency can be installed using the following command:

```
npm install --save react-test-renderer
```

STEP 2:

The snapshot testing for any component (here, the App component is being

tested) can be done as the following:

```
import React from 'react';
import TestRenderer from 'react-test-renderer';
import App from './App';
describe("App", () => {
 it("should render App component", () => {
  const AppComponent = TestRenderer.create(<App /
  >).toJSON();
  expect(AppComponent).toMatchSnapshot();
 });
});
```

`toMatchSnapshot` performs the snapshot testing by following the given steps:

1. If no snapshot exists, create a snapshot.
2. If a snapshot exists, the new snapshot will be compared with the last saved one.

STEP 3:

To run the test cases, execute the following command:

```
npm test
```

The snapshot will be created and saved in `__snapshots__` folder as the `App.test.js.snap` file.

STEP 4:

Now, if you make any changes to the App render output, the test case will fail showing you the changes done with respect to the old snapshot with the given failed message as shown in the following screenshot:

Figure 17.10: *Snapshot testing error*

You can rerun with the u option and it will update the snapshot with the latest changes after which your test will pass.

With renderer, the entire DOM structure of the components is rendered.

Shallow and mount

Shallow, mount, and render are the different approaches of rendering the component for the purpose of testing using enzyme.

- With the shallow approach, only the component is considered for testing without getting into the details of rendering the child components. It has access to the lifecycle methods for testing. This is

ideal for unit testing as it does not bother about the structure of the children but only the component being tested.

- With the mount approach, the entire component structure is rendered, including the child components and the lifecycle methods like `componentDidMount` and `componentDidUpdate` can be tested.
- With the render approach, only the render function is called and the component is rendered with its child components.

For snapshot testing, the three approaches can be used as follows:

```
import React from 'react';
import {shallow, mount, render} from 'enzyme';
import App from './App';
import { configure } from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';

configure({ adapter: new Adapter() });
describe("App", () => {
 it("should render App component with shallow", () => {
  const AppComponent = shallow(<App />);
  expect(AppComponent).toMatchSnapshot();
 });
 it("should render App component with mount", () => {
  const AppComponent = mount(<App />);
  expect(AppComponent).toMatchSnapshot();
 });

 it("should render App component with render", () => {
  const AppComponent = render(<App />);
  expect(AppComponent).toMatchSnapshot();
 });
});
```

If the child component structure changes, the mount and render will fail, but the shallow test case will not fail unless the component itself changes.

Starting with comparing snapshots to detect the changes in the component layout, the complexity of testing will increase depending on the complexity of the component and its functionalities.

Using Enzyme and Jest, all the component functionalities like prop and state changes, user actions like button click, input entered, etc. can be automated and tested. Following are some code snippets for the test scenarios:

Sample test scenario	Testing logic
To check or manipulate the value of a prop	`expect(wrapper.prop(props.children).toEqual('property value`

```
to check');
expect(AppComponent.state.input).toEqual('Value to compare
with');
});
it('has a specific DOM element in the component
   AppComponent= shallow(<AppComponent />)
   expect(AppComponent.find('h3')).toHaveLength(1);
    });
it('has a title of React Rocks!, () => {
    AppComponent= mount(<AppComponent />)
// mount used to check the child components also
    expect(AppComponent.find('.title').text()).toBe("React
    Rocks!")
 })
```

To check the value of the state

To find a specific DOM element in the component

Table 17.1: *Some test scenarios with sample code logic*

Automation testing is a much elaborate topic, but it is a topic worth exploring and incorporating in your application to reap the benefits in the long run. With this introduction of basic setup and usage, you should be able to explore this further.

Deploying React applications

After having tested well and ensured that our application is good to be delivered or shared for the user community to use, the next step is to deploy the application, so that it is available and accessible on a shareable link for people to use it.

'Deploying an application' means taking it to the production environment where it can be used by real users of the application. As a part of this process, our application is converted into a set of files which serve as the application over the network. For single page applications like the ones built using React, the set of files which get generated for deployment include:

- A single HTML file, index.html
- A set of single or multiple JavaScript files which include the CSS content for styling
- Static assets like images/icons / svgs

In order to deliver a highly performant application, we should make the application load faster by having a smaller file load size and removing any unnecessary part of this load.

As we used create-react-app to create our application template, it already has webpack configured as the bundler.

By running the following command, the deployment-related files will be generated in a folder named build:

```
npm run build
```

This command will optimally bundle and minify the code to have the best possible bundle size of the application.

You will receive a warning during the build process if you have any wrong or unused code, which you should correct before going for the final build.

When the operation completes, you can see the details of what happened along with the warnings and how you can proceed with deployment further as shown in the following screenshot:

Figure 17.11: *Instructions after the npm run build command with code details generated for deployment*

Once the minified build files are generated as shown, you can proceed with the deployment process as explained in *Chapter 12: Build and Deploy an Application*.

Conclusion

In this chapter, we learned about the React development world with respect to tools used for debugging, testing, and deploying single page application built using React.

We learned about the React and Redux developer tools which help in debugging React applications in a convenient way. We also learnt about the automation testing and process of deploying React applications. We are nearing the end of this book and the beginning of a new phase for you.

In the next chapter, which is also the last chapter of this book, we will discuss the additional skills which we need to pick up to further empower ourself to excel as we take on this web development path.

Questions

1. The ___ function is used for snapshot testing in Jest.

   A. toMatchSnapshot
   B. checkSnapshor
   C. diffSnapshot
   D. None of the above

   Answer: A

   `expect(AppComponent).toMatchSnapshot();`

   This command is used to perform snapshot testing on the given component.

2. Which of these are valid test file conventions to be picked by a test runner?

A. Files with `.js suffix in __tests__ folders.`

B. Files with `.test.js suffix.`

C. Files with `.spec.js suffix.`

D. All of the above

Answer: D

All the above conventions can be used.

3. With the ___ testing approach, only the component is considered for testing without getting into the details of rendering the child components.

A. mount

B. render

C. shallow

D. deep

Answer: C

In the shallow approach, only the component being tested is rendered without getting into the details of rendering the child components.

4. Which of these cannot be determined using Redux devtools while debugging your React-Redux application?

A. If the action got dispatched and with a proper type and payload, in the correct order.

B. If the store got initialized correctly.

C. If the store got updated correctly as expected.

D. If the prop value got passed and changed correctly.

Answer: D

The value of Props can be checked using the React devtools. Props should not be changed within the component.

5. The ___ represents the current active node being inspected which can be rendered in the console.log using this representation.

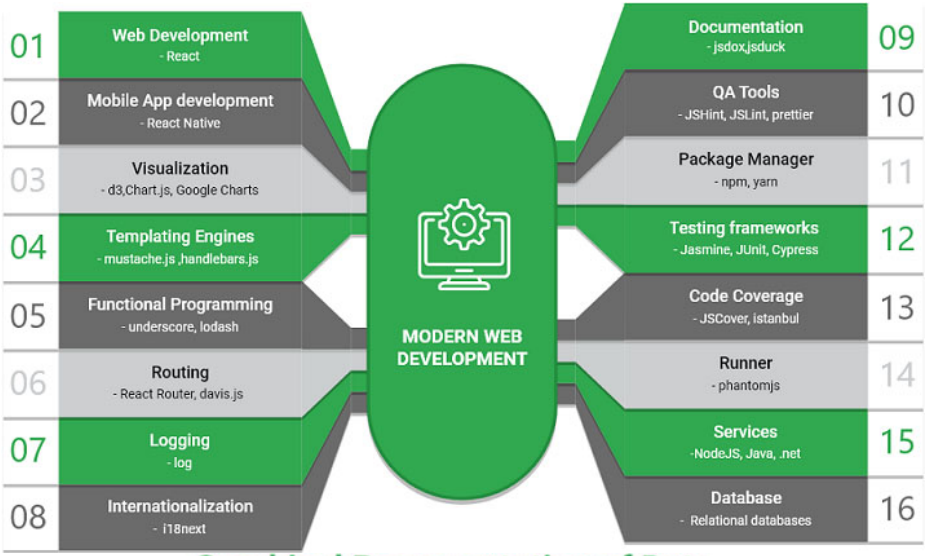
A. $c

B. $d

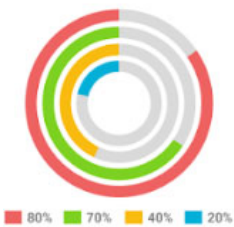
C. $r

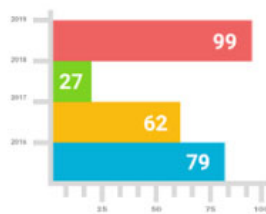
D. None of the above

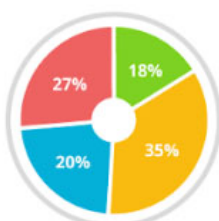
Answer: C

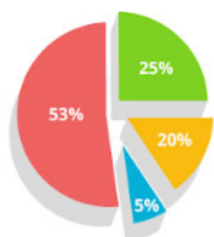


Graphical Representation of Data

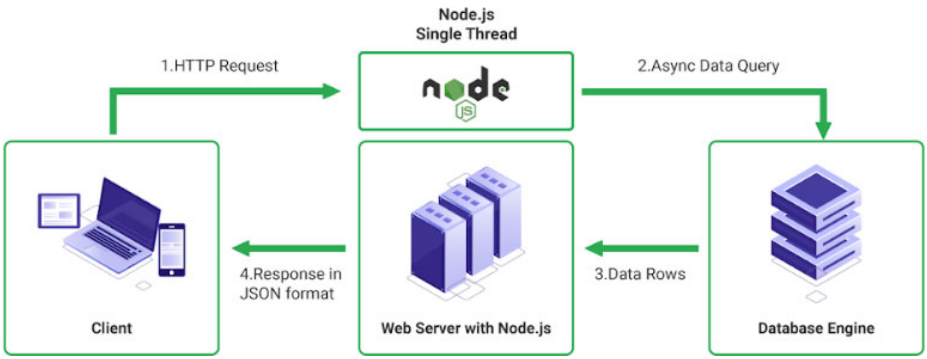


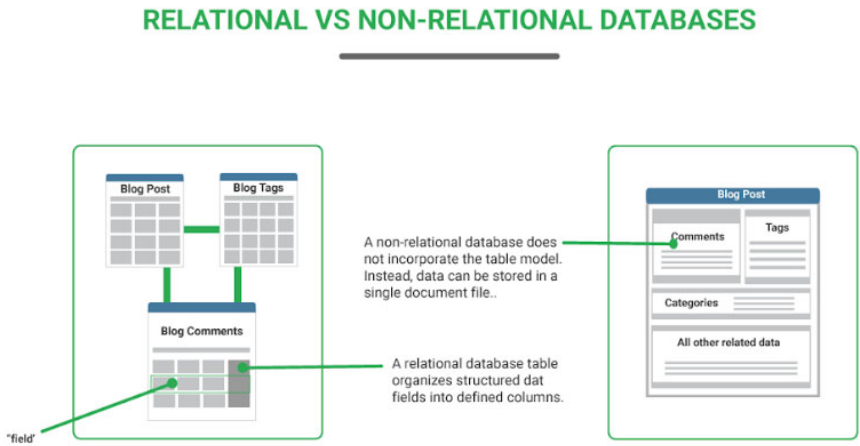






react-router	• Core Navigation Library, included in the other libraries
react-router-dom	• Navigation for dom based apps
react-router-native	• Navigation for react-native apps




RELATIONAL VS NON-RELATIONAL DATABASES
Blog Post
Blog Tags
Blog Comments
"field"
A non-relational database does not incorporate the table model. Instead, data can be stored in a single document file..
A relational database table organizes structured dat fields into defined columns.
Blog Post
Comments
Tags
Categories
All other related data

CHAPTER 18

What Next - For Becoming A Pro?

“Have a great aim in life, continuously acquire the knowledge, work hard and persevere to realize the great achievement.”

—Dr. A.P.J. Abdul Kalam

“The limit of your present understanding is not the limit of your possibilities.”

—Guy Finley

You are now in the last chapter of this book. In this book, you were introduced to a number of technologies, namely, HTML, CSS, JavaScript, and React.

Powered with this knowledge, in this chapter, you will be presented with some other skills related to web development which you can take up next and learn to enhance your learning. These skills will help you understand the full stack and move to the next level of this web development game. It also gives a peek into the different aspects of application development.

Structure

- Modern web development with JavaScript frameworks
- Building mobile apps
- Templating engines
- Visualization
- Functional programming
- Routing
- Logging
- Internationalization
- Documentation
- QA tools
- Package manager
- Testing frameworks
- Code coverage
- Runner
- Services
- Database

Objective

At the end of this chapter, you will cover the following topics:

- Be able to understand the different aspects of web development and the different technologies and skills associated with each of them
- Be briefly introduced to each of these skills which you can take up in detail and improve your ability to deliver the production-ready applications

Modern web development with JavaScript frameworks

The world of JavaScript and its frameworks is really huge and it continues to grow with new frameworks and packages being developed and

introduced to the world.

The following diagram summarizes some of the popular tools used in modern web development that you will come across in this chapter:

Figure 18.1: *Modern web development with JavaScript frameworks*

Building mobile apps

In this book, we dived into the world of web development. There is another parallel and equally enticing world of mobile app development. React Native is for mobile application development as React is for web development.

React Native is another powerful open source JavaScript (JSX) framework for mobile app development also created by Facebook. With the knowledge of React, it is easier to pick up React Native due to some common concepts like props; state, JSX, and components. React Native makes use of built-in Native components instead of React's web components.

React Native is a cross-platform solution with good performance which can be compared to the likes of iOS and Android Native apps as they are native and make use of JavaScript components that are built on iOS or Android components. The underlying development principle is the same and can be reused for different platforms. Platform-specific code can also be added based on the requirements of different styling or behavior.

Coming from a web background, you can get started with React Native using the Expo CLI, without having to install and configure Xcode or Android Studio. The other more native approach is using the React Native CLI, which requires Xcode or Android Studio as the pre configurations to get started.

React Native is becoming widespread and popular as it speeds up the overall mobile app development process with the benefits of near native app development. It is also easier to learn for a person coming from a web development background due to its similarity with web.

As a `React.js` developer, if you want to try your hands-on mobile app development, this could be your next step.

Templating engines

Templating engines are used to perform string interpolation. The purpose of templating engine is to make designing HTML pages easier by including interpolations in the template files. At runtime, the template engine will replace the interpolation variables in the template file with actual values and convert the template into a proper HTML file which is sent to the browser to be rendered.

Some of the popular templating engines include `mustache.js` and handlebars.js. mustache.js, as JavaScript implementation of Mustache provides the logic-less template syntax consisting of tags, where it can contain anything, including HTML, config file, and source code. Similarly, `handlebars.js` is a templating engine; however, it is based on Mustache and it further improves Mustache by adding the functionalities like support for the logic tags, helpers, paths, inverted sections, and so on.

Visualization

Data visualization is graphical representation of data. By means of graphs, charts, and pictures, data becomes more comprehensible and readable. It becomes easier to identify trends or variations in data. Huge volumes of data can be aggregated and presented in a visually understandable format using graphs and charts. Data is an important aspect in today's world where everything is driven by data. Hence, visualization becomes a key-ask in application development to manage and present readable data.

The following diagram shows a variety of graphs and charts which present data in a visually appealing format:

Figure 18.2: *Data visualization with graphs and charts*

There are many JavaScript libraries which enable data visualization; some of which include:

- d3.js: One of the most popular and powerful JavaScript visualization libraries for HTML and SVG.
- `Charts.js`: One of the simpler and flexible JavaScript visualization libraries.
- `Google Charts`: Another popular JavaScript visualization tool for drawing customizable charts.

Data, being so important and voluminous in today's world, being able to present data in a visually impactful representation using these visualization tools to its full potential, will surely be an important skill to master. Specifically, when the transactional users also demand a great dashboard and intelligent data visualization, the tools like D3.js play a key role.

Functional programming

Functional programming is a programming pattern which focuses on developing programs using pure functions to provide specific functionalities without having any shared state and mutable data. Functional programming JavaScript libraries are used to extend JavaScript's capabilities and provide functionalities which can be used

without replicating the implementation logic.

Some of the common and very useful libraries include underscore and lodash. They provide you a number of useful functions (for example, `map`, `reduce`, `find`, `filter`, and so o.) without extending any existing JavaScript object.

Routing

In the **Single Page Applications (SPA),** there is a single `index.html` which gets loaded based on the underlying JavaScript framework logic used. So, by default, a SPA will have only a single URL which will not change as we navigate in the application. In order to achieve a changing URL behavior and have functionality-specific URL path, we need to implement routing in our application. Routing allows the SPA to navigate through different URL paths and provide optimal experience to the end user.

React Router is the routing library for React and it is the collection of navigational components, which enable navigation across different URLs in applications and also to navigate forward, backward, and restore the state without reloading the complete application.

For web applications, `react-router-dom` is the application dependency to be installed in order to enable routing in the application. For mobile applications, react-router-native is the required dependency to enable routing.

The following diagram summarizes the different React packages used for routing in React applications:

Figure 18.3: *React Router libraries*

You can enable routing in your applications using the appropriate libraries.

Logging

Logging is a basic programming step where logs are added to the code to help understand the flow as well as aid in debugging errors. It helps during development to understand how the code works. In case of live projects, it helps in auditing and investigating issues during runtime.

Logging level can be set up to show minimal to more logs depending on if it is required during development or after deployment.

Instead of using the humble and most common console.log, we have some other tools and libraries to make logging more structured. Logging libraries like logs enable styling of the console logs to give a better visual representation of the logs. Console is a debug panel which provides a wrapper around the basic JavaScript console functionality and displays in

a controlled manner. The development process is empowered by the use of these logging libraries.

Internationalization

Technology has made the world a smaller place making extreme ends of the globe accessible and reachable to each other. Anyone residing anywhere on the globe may be interested in knowing and reading your work in spite of not knowing your language. This creates the requirement for localized apps which are adaptable to the needs of local languages and formats.

As per **whatis.com**, internationalization is the process of planning and implementing products and services so that they can easily be adapted to specific local languages and cultures, a process called localization.

Internationalization and localization have become a common requirement in today's applications to increase the global reach of the content beyond boundaries.

I18next is one of the most popular and well-established JavaScript frameworks which provide complete internationalization-related functionality requirements. React-i18next is another package built using the i18next ecosystem and making it usable in React applications. Another popular library used for internationalization enablement in React applications is react-intl.

These libraries help you to manage your translations, handle region-specific format related requirements, for example, date or number formats. They do not perform the translations for you but provide a well-defined infrastructure to handle translations which you can use and set up your required language translations.

Documentation

Documentation is another important aspect in programming to make your code clear and easily understandable. There are many JavaScript document generator libraries which make the task easier for you by generating supporting documentation for your code. Some of such document generator libraries include `jsdoc`, `jsduck`, `doc`, `documentation.js`, and so on.

Testing frameworks

In the *Chapter 11, Automation Unit Testing*, you were introduced to the concept of unit testing and how it can be done using Jasmine. Unit testing is testing a component independently. There are other types of testing like **E2E (end-to-end),** integration testing, and so on which look at testing the

entire application.

Cypress is a fast and fully functional end-to-end testing framework which includes assertions, mocking as well as stubbing capabilities to provide a perfect platform to automate end-to-end test scenarios.

QA tools

In this book, you became familiar with some testing and some test tools. However, another set of **Quality Assurance (QA)** tools which ensure good code quality by enforcement through automated checking include the linting tools.

Linting is an automated process of checking code for common programmatic errors and style guide deviations.

Being an automated process, it improves productivity by pointing out these issues upfront without the need for manual review and verification and hence, ensures better enforcement of good coding practices.

JSLint is one of the oldest linting tools for detecting common issues in the JavaScript code. It has a predefined configuration based on the good parts of JavaScript and cannot be modified or extended.

JSHint is a more flexible and configurable extension of JSLint that reports any improper use of variables like global variables, unused variables, and statements like missing return, break in a switch statement in programs written in JavaScript. Apart from the default behavior, you can also configure these rules based on your needs, which makes it much more flexible to incorporate.

ESLint, the newest and most popular in this group, makes use of the ES6 syntax to impose the coding styles and detect common problems with the code.

Package manager

Package managers host the different open source JavaScript libraries and provide tools for fetching and packaging them as dependencies for any application development.

We have already used the **node package manager (npm)** for installing our dependencies in our example applications. Yarn is another package manager picking up on popularity due to its speed, reliability, and security-related additional capabilities.

Code coverage

Code coverage is a measure of the percentage of code which is covered by automated tests. It determines which statements in a body of code have

been executed through a test run, and which statements have not. For computing code coverage in JavaScript, there are some libraries available, namely, JSCover and istanbul. Specifically, with a strong focus on **Continuous Integration (CI)** and **Continuous Deployment (CD),** the importance of code coverage has become very high and it indicates your preparedness for the CI/CD process.

Runner

Runner provides the capability to run the application without a **graphical user interface (GUI)** and a normal web browser. This is useful to automate web testing and network monitoring.

`Phantom.js` is the most popular in this category and it is described as a **Scriptable Headless Browser**. It is a browser without a GUI. It can be used to programmatically launch the application, perform automations to access some data or screen shots from the application, run automated functional test cases with frameworks such as Jasmine and monitor the network and application performance by automating performance testing.

Services

The definition of services from webopedia.com states, *The term Web services describes a standardized way of integrating Web-based applications using the XML, SOAP, WSDL and UDDI open standards over an Internet protocol backbone.*

Web services enable sharing of data and logic between different applications using an **application programming interface (API)** across a network without exposing its complete details to each other. Web services allow different applications built using totally different technologies to be able to communicate without the need of any complex interfaces.

Web services can be built using Node.js, the JavaScript runtime engine, which is used for server-side programming using JavaScript. Java and ASP.NET also provide their own APIs to create standard web services to communicate and share data between diverse applications.

The following diagram depicts the flow of request response to and from a Node.js server:

Figure 18.4: *NodeJS Services Data flow*

Knowing how to build and use web services and microservices will enable you to visualize your application end-to-end and provide the best experience for the end users.

Database

“Data is a precious thing and will last longer than the systems themselves.”

– Tim Berners-Lee, inventor of the World Wide Web.

Every application has some amount of data, which holds the information related to the application. The following image shows the difference of structure in the two types of databases: relational and non-relational:

Figure 18.5: *Relational and non-relational databases*

Database is a system which enables you to store, access, update, and manipulate the data in a structured way. It is an organized collection of information. Relational databases comprise sets of data segregated into tables and organized as rows and columns of data. Also, there exist well-defined relationships and associations between the different tables holding data. To work with relational databases, we use a structured query language like SQL to fetch and manipulate the data.

Examples of relational databases include Oracle, MySQL, Microsoft SQL Server, PostgreSQL, and IBM DB2.

Non-relational databases do not follow the table, row and column structure like relational databases. Instead they focus on storing data in the most optimized way depending on the structure of the data. They may exist as records with key-value pairs or plain JSON objects. They are also referred as No-SQL as they do not use SQL.

Examples of No-SQL databases include MongoDB, DocumentDB, Cassandra, Coachbase, HBase, Redis, and Neo4j.

Conclusion

In this chapter you read about different aspects of web development and the different tools and technologies associated with each of them.

Having been introduced to all these new skills related to the world of modern web development with JavaScript, you can now start to explore each of these further and enhance your skill sets. This is only the beginning! Follow best practices and coding guidelines in your work, keep reading and learning something new, every single day, and you will turn into a talented professional, with whom anyone would like to be associated with.

You are now ready to get going in the web development world! Bring it on!

Questions

1. Non-relational databases comprise sets of data segregated into tables

and organized as rows and columns of data.

A. True

B. False

Answer: Option B

2. Which of the following is a relational database?

A. Cassandra

B. Coachbase

C. HBase

D. PostgreSQL

Answer: Option D

3. ____ is an automated process of checking code for common programmatic errors and style guide deviations.

A. Lexical verification

B. Linting

C. Code coverage

D. Visualization

Answer: Option B

4. Which of the following is a JavaScript library which enables data visualization?

A. d3

B. Charts

C. Google charts

D. All of the above

Answer: Option D

5. ___ is a fast and fully functional end-to-end testing framework which includes assertions, mocking as well as stubbing capabilities to provide a perfect platform to automate end to end test scenarios.

A. Cypress

B. Mustache

C. Handlebars

D. All of the above

Answer: Option A